Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Automatické ověřování podpisů

2018

Bc. Lukáš Oščádal

Vedoucí práce: RNDr. Eduard Bartl, Ph.D.

Studijní obor: Informatika, prezenční forma

Bibliografické údaje

Autor:	Bc. Lukáš Oščádal
Název práce:	Automatické ověřování podpisů
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2018
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	RNDr. Eduard Bartl, Ph.D.
Počet stran:	52
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Bc. Lukáš Oščádal
Title:	Automatic signature testing
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2018
Study field:	Computer Science, full-time form
Supervisor:	RNDr. Eduard Bartl, Ph.D.
Page count:	52
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Ověření podpisu pomocí konstrukce jeho kostry a měření míry její deformace při formování na vzorový podpis je hlavním tématem této práce. Snahou je rozvinout tento přístup s využitím různých typů fuzzy inferenčního vyhodnocování a principů převzatých ze segmentace obrazu pomocí aktivních kontur. Algoritmus deformace pracuje s kostrou podpisu jako s tvarově paměťovým materiálem s fyzikálními vlastnostmi ohebnosti a pružnosti.

Synopsis

The main topic of this thesis is signature veryfying based on constructing the signature skeleton and measuring its deformation during the transition to original signature. The aim is to develop the method using different types of fuzzy inference and principles taken from image segmentation. The deformation algorithm works with the signature skeleton as if it had physical properties of formability and elasticity.

Klíčová slova: aktivní kontury, fuzzy logika, ověření podpisu

Keywords: active contours, fuzzy logic, signature veryfying

Děkuji panu RNDr. Eduardu Bartlovi, Ph.D za jeho velmi cenné konzultace a věcné připomínky, bez kterých by tato práce nezvnikla.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

1 Úvod

Tato práce se zabývá problémem ověření pravosti ručně psaného podpisu na základě jediného vzorového podpisu a to ve formě bitmapových souborů. Pro ověření tedy nejsou k dispozici informace o přítlaku pera, rychlosti psaní a podobně. Práce staví na myšlenkách prezentovaných ve článku *Fuzzy shape-memory snakes for the automatic off-line signature verification problem* [3] z roku 2008, který vyšel v časopise *Fuzzy sets and systems*. Autoři zde popisují možný přístup ověřování podpisů za pomocí tvarově paměťových hadů (aktivních kontur) a fuzzy inferenčních mechanismů. Hlavním cílem byla implementace, rozšíření a případná modifikace těchto přístupů pro ověřování podpisů.

V následujícím textu je podrobně rozebrána problematika přístupu ověřování a celý soubor algoritmů k tomuto účelu vytvořených. V kapitole č. 2 jsou stručně shrnuty teoretické znalosti, které jsou nezbytné pro pochopení dalšího textu. Čtenář bude v krátkosti uveden do teorie fuzzy množin a fuzzy inferenčních mechanismů na ně navazujících. Dále jsou zde popsány základy tvarově paměťových hadů, kteří tvoří jádro principu popisovaného přístupu ověřování.

Kapitola č. 3 se zabývá algoritmy, které byly použity při implementaci výstupní aplikace této práce. Popisuje problémy, které tyto algoritmy řeší a porovnává je s naivními přístupy, které nejsou pro řešení vhodné.

Další kapitola se zabývá implementací a architekturou celé aplikace. Popisuje a znázorňuje logické uspořádání a význam částí aplikace. Vysvětluje reprezentaci jednotlivých typů dat a principy práce s nimi na úrovni implementace.

Kapitola č. 5 popisuje systémové požadavky a instalaci aplikace včetně podrobné uživatelské příručky. Poslední kapitola pak shrhuje výsledky, které byly zjištěny během testování.

2 Teoretický úvod

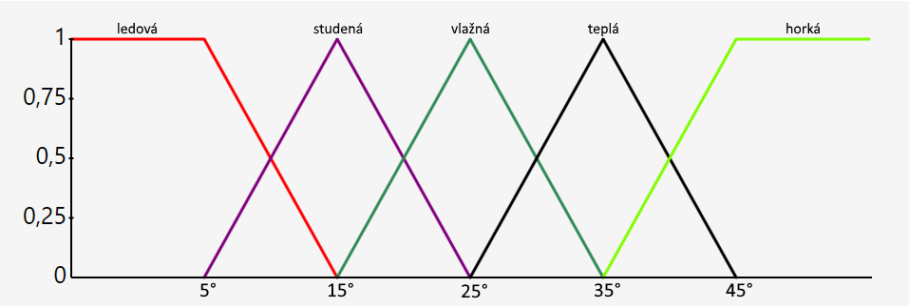
Tato kapitola obsahuje stručné shrnutí teoretických znalostí, které jsou nezbytným základem pro pochopení zbytku této práce. Popisuje základy teorie fuzzy množin a fuzzy logiky z ní vycházející. Dále popisuje princip a aplikaci fuzzy inferenčních systémů, které jsou v této práci použité. Druhým tématem této kapitoly jsou tvarově paměťoví hadi, pomocí kterých budou následně reprezentovány testované podpisy a s jejichž pomocí bude prováděno celkové vyhodnocení ověřování.

2.1 Fuzzy množiny

Pojem *fuzzy množina* popisuje množinu, která nemá ostře stanovené hranice a příslušnost prvků k této množině je odstupňovaná. To znamená, že každý prvek fuzzy množiny má přiřazené reálné číslo z intervalu $\langle 0, 1 \rangle$ vyjadřující stupeň, v jakém patří do této množiny. Prvky, které do množiny patří úplně, mají stupeň příslušnosti 1 a prvky, které do množiny nepatří vůbec mají stupeň příslušnosti 0.

Kromě toho však můžou existovat prvky, které do množiny patří jen částečně a to například ve stupni 0,5.

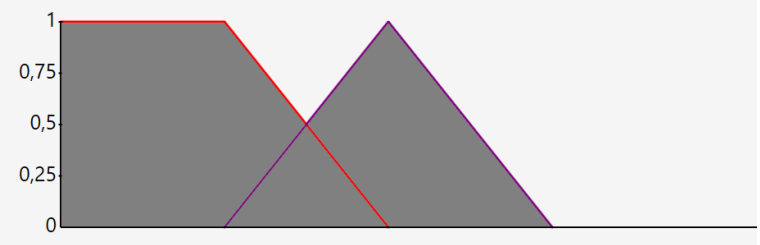
Jako příklad lze uvést třeba množinu vysokých lidí. Lidé s výškou 160 cm budou mít stupeň příslušnosti 0, lidé vysocí 175 cm budou mít stupeň příslušnosti 0,4 a lidé vysocí 200 cm budou mít stupeň příslušnosti 1. Určení stupňů příslušnosti k daným fuzzy množinám bývá záležitostí expertů daných problematik a tyto informace jsou vstupem pro stanovení fuzzy množin. Na obrázku č. 1 jsou znázorněné fuzzy množiny vyjadřující stupně teploty vody.



Obrázek 1: Fuzzy množiny vyjadřující teplotu vody

2.1.1 Operace nad fuzzy množinami

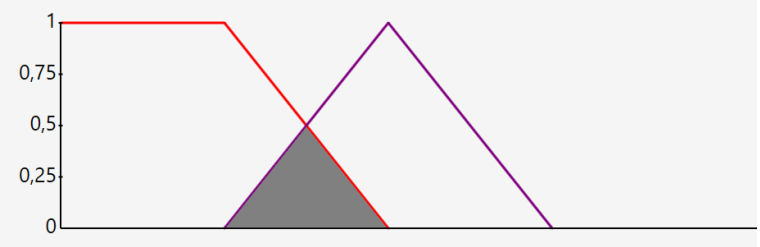
Stejně jako s klasickými množinami, tak i s fuzzy množinami lze provádět některé zakladní množinové operace, jako jsou: *průnik*, *sjednocení* a *doplněk*. Tyto operace jsou znázorněny na obrázku č.2, 3 a 4.



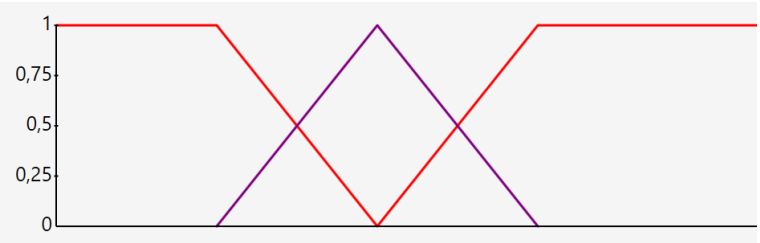
Obrázek 2: Sjednocení fuzzy množin

2.1.2 Fuzzy inferenční systém

Inferenční systémy se využívají jako vyhodnocovací jednotka uvnitř řídících a regulačních systémů. Inferenční systém je obecně řečeno způsob stanovování hodnot výstupů na základě vstupních hodnot vůči pravidlům inferenčního systému. Tyto systémy se skládají z množiny inferenčních pravidel a báze dat, kterou využívá systém při vyhodnocování pravidel. Obecný postup, jakým inferenční systém pracuje, je, že nejprve ze vstupu načte a zpracuje hodnoty vstupních veličin, poté



Obrázek 3: Průnik fuzzy množin



Obrázek 4: Doplněk fuzzy množiny

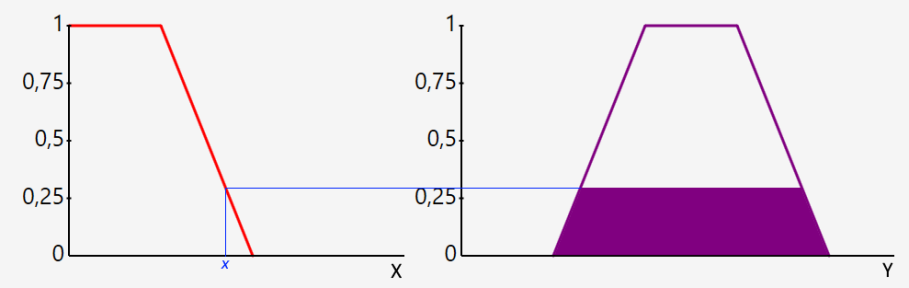
vyhodnotí každé inferenční pravidlo vůči těmto vstupním hodnotám a výsledky všech pravidel zkombinuje. Posledním krokem je určení ostré hodnoty výstupní veličiny z kombinace výsledků jednotlivých pravidel a zapsání této hodnoty na výstup.

Fuzzy inferenční systém je specifický tím, že jeho pravidla jsou tvořena fuzzy množinami a báze dat obsahuje definice těcho fuzzy množin pro všechny vstupní i výstupní veličiny. Fuzzy inferenční pravidla bývají nejčastěji ve tvaru

$$\text{Antecedent} \implies \text{Konsekvent},$$

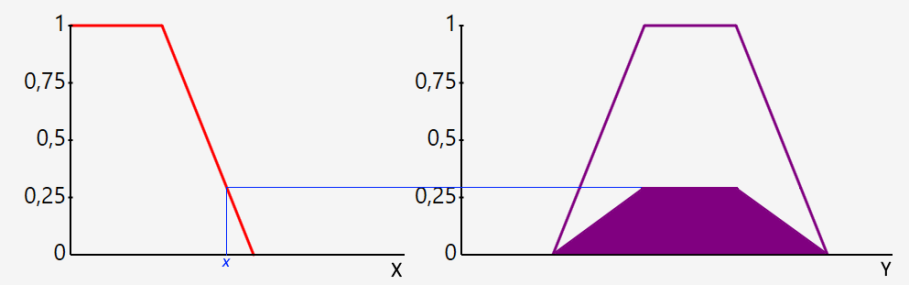
kde *Antecedent* je množina fuzzy množin příslušných ke vstupním veličinám a *Konsekvent* je množina fuzzy množin příslušných k výstupním veličinám. Význam inferenčního pravidla lze tedy jednoduše popsat tak, že pokud vstupní veličiny nabývají hodnot obsažených v antecedentu, pak výstupním veličinám budou přiřazeny hodnoty z konsekventu. Konkrétněji pro fuzzy inferenční pravidla lze říci, že čím většího stupně příslušnosti nabývají vstupní veličiny k fuzzy množinám v antecedentu (čím více splňují antecedent), tím blíže budou hodnoty výstupních veličin stanoveny k hodnotám v konsekventu. Na obrázku č. 5 je znázorněné fuzzy inferenční pravidlo pro vstupní veličinu X, výstupní veličinu Y a hodnotu x vstupní veličiny.

Fuzzy inferenčních systémů jako takových je obecně více. Nejčastěji se liší způsobem vyhodnocování jednotlivých inferenčních pravidel a stanovením ostré výstupní hodnoty z kombinace výsledků vyhodnocení pravidel. Jedním z častých způsobů vyhodnocování pravidel je způsob, kde výsledkem vyhodnocení je fuzzy množina konsekventu „oříznuta“ na stupeň, v jakém je splněn antecedent pravidla tak, jak je to znázorněno na obrázku 5. Druhým používaným způsobem je vyhodnocení, jehož výsledkem je opět fuzzy množina konsekventu, jejíž stupně



Obrázek 5: Fuzzy inferenční pravidlo a jeho vyhodnocení

příslušnosti jsou vynásobeny stupněm příslušnosti, v jakém je splněn antecedent pravidla. Tento způsob vyhodnocení znázorňuje obrázek č. 6.



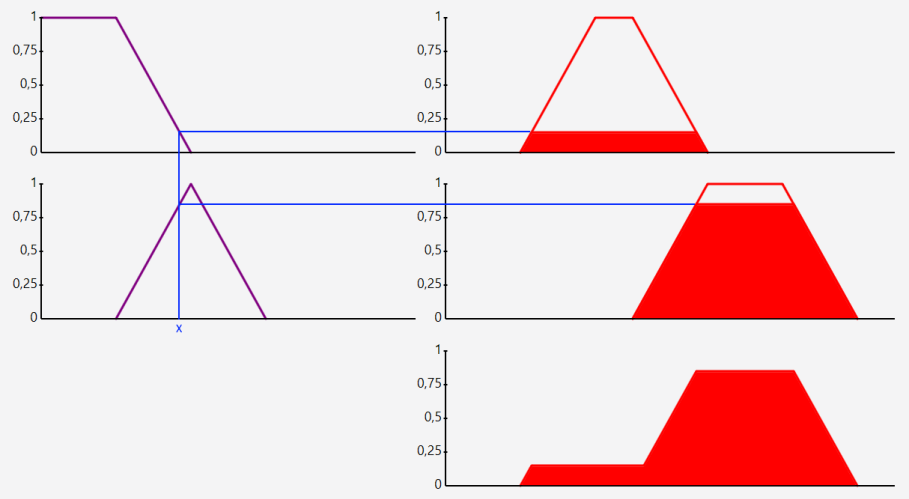
Obrázek 6: Fuzzy inferenční pravidlo a jeho vyhodnocení

Pro získání kombinace výsledků všech inferenčních pravidel se používá operace sjednocení fuzzy množin. Takto vzniklá kombinace je tedy opět fuzzy množina a je příslušná k výstupní veličině inferenčního systému. Obrázek č.7 ukazuje vyhodnocení dvou inferenčních pravidel a následné sjednocení výsledků.

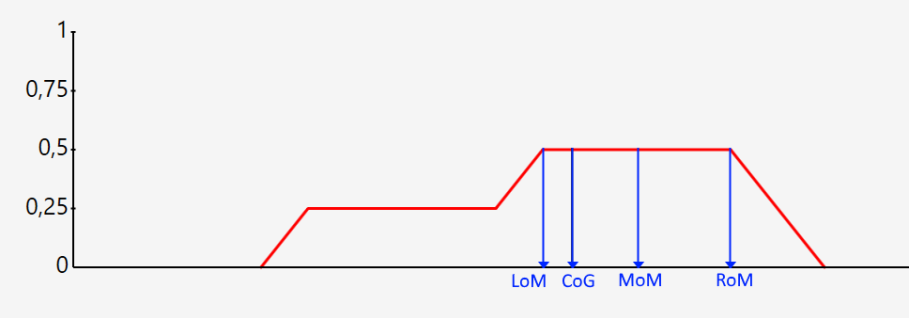
Pro stanovení ostré hodnoty výstupní veličiny z této fuzzy množiny se používá opět více způsobů, z nichž nejpoužívanější jsou *Center of gravity* (CoG), *Left of maximum* (LoM), *Right of maximum* (RoM), *Center of maximum* (CoM). Metoda *Center of gravity* využívá pro stanovení ostré hodnoty výpočet těžiště fuzzy množiny. Zbylé tři zmíněné metody určují ostrou hodnotu jako nejlevější, nejpravější nebo střední hodnotu, ve které dosahuje fuzzy množina nejvyššího stupně příslušnosti. Stanovení ostré hodnoty výstupní veličiny pomocí různých metod znázorňuje obrázek č. 8.

2.2 Aktivní kontury (hadi)

V počítačové disciplíně zpracování obrazu se vyskytuje úloha segmentace. Obecně se dá segmentace definovat jako rozdělení obrazu do podčástí odpovídajících nějakým objektům v obraze. Každý pixel obrazu má tedy přiřazen objekt nebo index objektu, do kterého spadá v rámci obrazu. Popsaných metod pro segmentaci je velké množství.



Obrázek 7: Kombinace výsledků pravidel



Obrázek 8: Metody stanovení ostré hodnoty výstupní veličiny fuzzy inferenčního systému

Jednou z nich je segmentace pomocí aktivních kontur (hadů). Tato metoda využívá kruhově uzavřený seznam bodů obklopující hranice objektu, který se iterativně smršťuje a přibližuje až na přesné hranice tohoto objektu. Tato deformace je řízena pomocí takzvaných vnitřních, obrazových a vnějších sil. Vnitřní síly ovlivňují hladkost průběhu kontury, jako jsou ohyb nebo zlom. Obrazové síly směřují deformaci směrem k hranám objektu a vnější síly jsou výsledkem počátečního umístění kontury na obraze. Tyto síly definují takzvanou energii kontury (hada), přičemž metoda segmentace pomocí aktivních kontur je založena na minimalizaci této energie. Způsobů, jakými lze stanovit přesnou hodnotu energie hada, je mnoho. Například pro hada $h = p_1, p_2, p_3, ..., p_n$, kde $p_i = [x_i, y_i]$ je i–tý bod hada, může definovat energii jako

$$E_h = \sum_{n=1}^{N} E_{int}(p_n) + \sum_{n=1}^{N} E_{pic}(p_n) + \sum_{n=1}^{N} E_{ext}(p_n).$$

Pomocí hadů a definic jejich energií lze vytvořit takzvaného tvarově pamětového hada, který si pamatuje svůj původní tvar před deformací. V průběhu

iterativních deformací pak lze měřit míru deformace hada pomocí jeho celkové energie. Tato energie vzroste vždy, když dojde k deformaci úhlů mezi jednotlivými body hada nebo když dojde k deformaci segmentu (spojnice mezi dvěma body hada). Deformací segmetu se rozumí stlačení nebo natažení tohoto segmentu, neboli změna původní vzdálenosti krajních bodů segmentu.

3 Algoritmy

Hlavní myšlenkou pro algoritmus ověřování podpisů v této práci, je využití tvarově paměťových hadů jako kostry testovaných podpisů. Ti jsou iterativně deformováni do tvaru vzorových podpisů (tedy podpisů, u kterých autora známe) a následně se vyhodnotí míra jejich deformace. Předpokladem je, že pokud je autorem obou podpisů stejný člověk, pak budou oba podpisy svým rozložením a tvarem podobné a tudíž míra deformace kostry jednoho podpisu na druhý bude malá. V opačném případě by se měly lišit tvarem linek a celkovým rozložením podpisu, což se projeví na míře deformace.

Takový přístup však zahrnuje několik problémů. Prvním problémem je postup, jakým deformovat kostru testovaného podpisu tak, aby jednotlivé body kostry měly tendenci konvergovat správným směrem ke vzorovému podpisu. Druhým důležitým aspektem je fakt, že algoritmus deformace by měl zohleňovat „tvarovou paměť“ kostry testovaného podpisu, aby výsledná deformace byla co nejvíce rovnoměrná a objektivní. To znamená, že nechceme aby deformace pohlcovala informaci o základních tvarových rysech testovaného podpisu.

Deformace kostry je prováděna postupně v iteracích a je zde využívána takzvaná *mapa vzdáleností*, která pro každý bod určí vzdálenost od nejbližšího bodu linky vzorového podpisu. Díky tomu lze docílit, aby každý bod kostry konvergoval směrem k lince podpisu. K tomu, aby si kostra stále zachovávala co nejvíce svůj původní tvar a deformace byla co nejvíce rovnoměrná, využívá algoritmus několik dílčích veličin. Na jejich základě algoritmus v každém kroku iterace vyhodnocuje nejlepší směr deformace.

V následujících kapitolách je podrobně popsáno, jakým způsobem probíhá celý proces ověření od počátečního zpracování bitmapových souborů, přes stavbu a deformaci kostry, až po celkové vyhodnocení.

3.1 Předzpracování bitmapy podpisů

Vstupem celé aplikace jsou bitmapové soubory testovaného a vzorového podpisu. Pro potřeby ověřování pravosti podpisů nejsou informace o odstínech pixelů nijak důležité a lze se omezit pouze na dvoubarevné spektrum a ostře tak rozlišit pixely podpisové linky od prázdného papíru. Omezením se na pouze dvě barvy, získáváme pro každý pixel binární informaci o tom, zda je či není pixelem podpisové linky. To umožňuje převést podpis do efektivní interní reprezentace pomocí dvourozměrného pole a binárního výčtového typu *Px:* { *bílý pixel, černý pixel*}.

Toto pole má stejné rozměry jako podpisový obraz a každý prvek obsahuje hodnotu výčtového typu Px, čímž reprezentuje pixel na stejných souřadnicích.

Předzpracování bitmapy podpisu probíhá ve dvou krocích. Nejprve je v paměti vytvořeno dvourozměrné pole o rozměrech bitmapy. Následně projdeme všechny pixely bitmapy a na příslušné místo v poli zapíšeme výčtový typ podle barvy pixelu.

Rozhodnutí o tom, zda bude pixel označen za bílý, nebo černý, provádí filtr, který pro každý pixel $p_{xy} : \langle r, g, b \rangle$ položí výčtový typ Px_{xy} jako:

$$Px(p) = \begin{cases} \text{bílý}, & \text{pokud } r \geq T \wedge g \geq T \wedge b \geq T, \\ \text{černý}, & \text{jinak}, \end{cases}$$

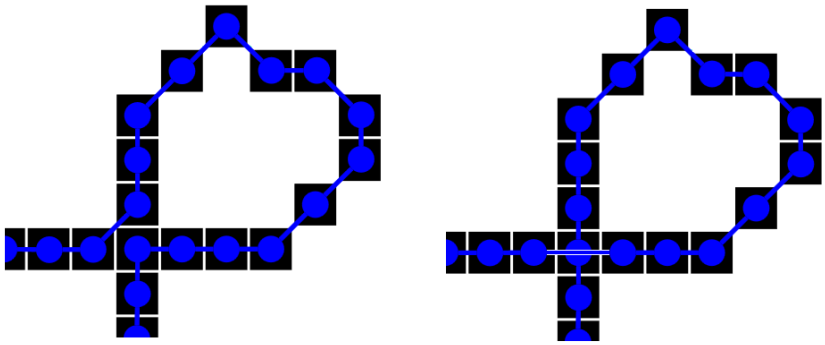
kde T je prahová hodnota, která byla testováním stanovena na hodnotu 220.

3.2 Stavba kostry podpisu

Jednou z hlavních myšlenek této práce byla reprezentace kostry podpisu pomocí tvarově paměťového hada popsaného v kapitole č. 2.2. Pokud však máme podpis pouze ve formě bitmapy bez dynamických informací o tom, jak podpis vznikal, tedy konkrétně kudy autor podpisu táhnul perem v závislosti na čase, pak je velice obtížné vyjádřit kostru lineární datovou strukturou s uspořádáním podle času vzniku daného bodu kostry.

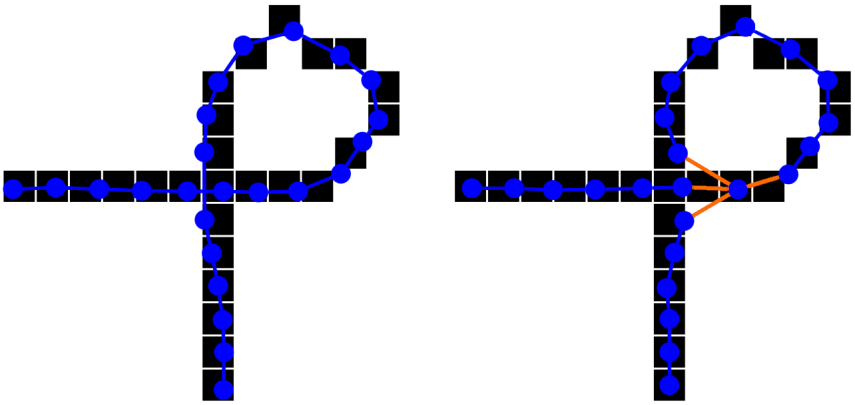
Předpokládejme pro jednoduchost, že linka podpisu je vždy jeden pixel široká a podpis je rozdělený do více linek (např. dvouslovný podpis nebo diakritika). Dále předpokládejme, že každý z těchto pixelů označíme jako bod kostry podpisu. K tomu, abychom kostru reprezentovali jako lineární seznam, potřebujeme znát uspořádání podle vzniku těchto bodů, které sice nemusí být úplné v rámci všech linek celého podpisu, ale musí být úplné v rámci jedné linky. Pro naše účely není důležitý směr tohoto uspořádání, jelikož je důležité pouze pro určení sousedů každého bodu v lineárním seznamu a tím i bodů, které budou mezi sebou propojeny jedním segmentem hada. Pokud je linka jednoduchá tak, že každý bod má maximálně dva sousedy, je úloha určení takového uspořádání jednoduchá. Problém však nastává u linek obsahujících jedno a více křížení. V takovém případě není jednoznačné jak tyto body uspořádat. Situaci křížení, při které vzniká nejednoznačnost ukazuje obrázek č. 9.

Kvůli tomuto problému byla pro reprezentaci kostry podpisu zvolena jiná hierarchická datová struktura vycházející z tvarově paměťového hada. Tato struktura je tvořena jednotlivými uzly, z nichž každý má uložený odkaz na každého svého souseda, se kterým tvoří koncové body segmentu kostry. Tím, že jsme umožnili, aby každý bod kostry mohl mít více než dva sousedy, jsme se odstínili od problému s uspořádáním. Nicméně na druhou stranu jsme do kostry vnesli nežádoucí vlastnost jakýchsi „pevných křižovatek“. Pojmem pevná křižovatka je myšlen bod, který vznikl právě křížením dvou linek a má tedy čtyři nebo více sousedů, se kterými je pevně spojen segmenty. Tyto body nejsou zcela žádoucí, protože jeho sousedé vznikali v různé momenty. Fakt, že se linky spojily právě



Obrázek 9: Nejednoznačnost uspořádání bodů hada

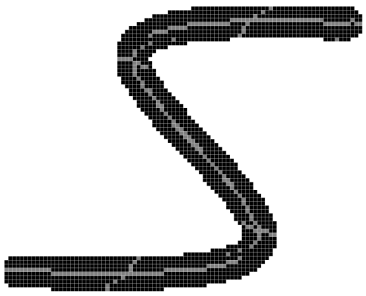
v tomto konkrétním bodě není zcela důležitý. Při deformaci na jiný byť pravý podpis se následně deformace projeví hned na několika segmentech právě kvůli tomuto pevnému spojení. Pokud bychom však měli kostru bez tohoto pevného spojení, míra deformace by byla podstatně příznivější tak, jak to znázorňuje obrázek č. 10.



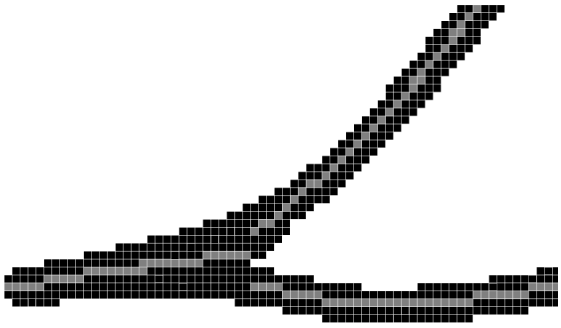
Obrázek 10: Pevné křižovatky kostry podpisu

Algoritmus konstrukce kostry

Aby kostra co nejlépe vystihovala tah ruky při samotném podpisu, je nutné, aby jednotlivé body kostry byly umístěny co nejvíce na středu linky. K tomuto účelu je nezbytné nalézt středovou linku všech linek podpisu o šířce jednoho pixelu. Jako první se nabízí naivní přístup, kdy v každém sloupci a každém řádku linky počítáme výšku a šířku linky. Jako střed pak bereme pixel uprostřed dané výšky nebo šířky. Takovýto přístup má však dva zásadní problémy. Jak ukazuje obrázek č. 11, každá vodorovná či svislá linka má vždy uprostřed své délky „špatně“ označené body, které jsou sice středem linky, ale v jiné ose než



Obrázek 11: První problém naivního hledání středu linky



Obrázek 12: Druhý problém naivního hledání středu linky

nás zajímá. Druhým zásadním problémem je situace, kde je linka velmi ostře zlomená nebo dochází k jejímu rozdvojení. V tomto případě nechceme, jen aby body kostry ležely na středu linky, ale aby kostra kopírovala původní tah pera, protože jak je vidět na obrázku č. 12, linie středů linky není v rozdvojení souvislá. Je evidentní, že ve sloupci pixelů, kde došlo ke spojení obou linek, bychom chtěli dva body, i když jsme na souvislém sloupci černých pixelů.

Algoritmus, který řeší problémy předchozího naivního přístupu, vychází z myšlenky rovnoměrného odebírání pixelů ze všech stran linky podpisu až na jednopixelovou linku. Takováto linka bude středovou linkou výchozího podpisu. Postupně tedy procházíme každý černý pixel a zjišťujeme, zda má bílého souseda. Pokud má bílého souseda pak, mohou nastat dvě situace:

1. Linka podpisu je v tomto místě silná alespoň dva pixely a odstraněním nepřerušíme linku. Pixel označíme jako bílý.

2. Linka je tenká 1 pixel a odstraněním by došlo k přerušení linky. Pixel ponecháme černý.

Odstraňování krajních pixelů se však musí provést až poté, co projdeme všechny pixely. Pokud bychom odstraňovali pixely ihned, v ten moment by se stali krajními pixely i sousedé odstraněného pixelu, kteří leží ve druhé vrstvě pod

tímto pixelem. Proto si při průchodu každou iterací pouze ukládáme souřadnice pixelů, které budou po skončení aktuální iterace odstraněny.

Jako mezikrok konstrukce kostry se ještě provádí filtrace této středové linky, aby se odstranily některé další zbytečné pixely. Tato filtrace se provádí pomocí shody se vzorovými maticemi pixelů.

Nyní tedy máme linku podpisu procházející středem původního podpisu a všechny body kostry by měly ležet na této lince. Našim cílem je vytvořit strukturu skládající se z jednotlivých uzlů, které znají své souřadnice a mají odkazy na všechny své sousedy, se kterými jsou spojené segmentem. Nejprve vytvoříme dvourozměrné pole uzlů o stejných rozměrech jako je bitmapa podpisu. Následně budeme procházet bitmapu středové linky a pro každý černý pixel vytvoříme uzel kostry, kterému nastavíme souřadnice daného pixelu. Na tyto souřednice ho také uložíme do pole uzlů. Poté budeme procházet toto pole a každému uzlu nastavíme odkazy na všechny uzly v sousedních kolonkách pole.

Tímto způsobem se vytvoří první „hrubá“ podoba kostry, která však zatím není ideální, jelikož obsahuje bod pro každý černý pixel středové linky podpisu a hustota bodů je tak velmi vysoká. Proto je nutné vytřídit nedůležité body, aby kostra podpisu obsahovala pouze body, které podpis skutečně vystihují. Při třídění musíme zachovát body, které:

1. Mají pouze jednoho souseda. Tyto body jsou počáteční a koncové body linek a definují, kde autor podpisu začal a kde skončil svůj tah.

2. Mají tři a více sousedů. Tyto body jsou průsečíky linek podpisu a jejich vynecháním by mezi uzly vznikaly vazby, které nedávají smysl.

Uzly, které mají právě dva sousedy, můžeme vynechat, ale je nutné zajistit, aby jsme tím nepřerušili kostru. Oběma sousedům se nejdříve zruší odkaz na aktuálně odstraňovaný uzel a následně se nastaví odkaz na druhého ze sousedů. Tímto způsobem se odstraní každý druhý uzel, krerý má právě dva sousedy. Tento postup se opakuje dokud nemá kostra podpisu požadovanou hustotu uzlů.

3.3 Výpočet mapy vzdáleností

Aby bylo možné nějakým způsobem řídit deformaci kostry testovaného podpisu na podpis vzorový, je nutné mít k dispozici způsob, kterým se bude algoritmus deformace orientovat na vzorovém podpise a byl tak schopen uzly kostry posouvat směrem k nejbližším linkám vzorového podpisu. K tomuto účelu slouží takzvaná „mapa vzdáleností“, která pro každou souřadnici bodu kostry určí, jaká je vzdálenost tohoto bodu od nejbližší linky vzorového podpisu. S těmito informacemi bude algoritmus deformace schopen vyhodnocovat „výhodnost“ jednotlivých posunutí bodů kostry a určit tak nejlepší směr a vzdálenost posunutí.

Opět nejdříve rozeberme naivní přístup a jeho problémy. Vezměme jako reprezentaci mapy vzdáleností dvourozměrné pole o rozměrech bitmapy podpisu, kde každé políčko obsahuje číslo vyjadřující vzdálenost k nejbližšímu černému

Obrázek 13: Mapa vzdáleností naivním algoritmem

pixelu vzorového podpisu. Na začátku nastavíme pro každý pixel bitmapy odpovídající kolonku pole tak, že pro každý černý pixel zapíšeme hodnotu 0 (tento bod leží na černém pixelu a proto je vzdálenost nulová) a pro každý bílý pixel nastavíme hodnotu -1 (vzdálenost zatím není známá). Nyní nastavíme číslo iterace i na 1 a začneme procházet každý nezáporný prvek pole a všechny jeho sousední prvky s hodnotou -1 nastavíme na hodnotu i. Potom, co projdeme všechny prvky pole, zvýšíme hodnotu i a začneme novou iteraci. Iterace se opakují, dokud nejsou všechny hodnoty v poli kladné. Výsledek tohoto algoritmu je zobrazen na obrázku č.13 pomocí odstupňování odstínů šedé barvy.

Jak lze vidět, algoritmus selhává pro vzdálenosti, které nejsou rovnoběžné s některou osou a vytváří tak nerovnoměrné hranaté vzory. To je způsobeno tím, že diagonálním sousedům je vždy nastavena vzdálenost 1 stejně jako vertikálním a horizontálním sousedům, což není pravda, protože diagonální sousedé leží ve vzdálenosti $\sqrt{2}$. Možnou modifikací je naopak procházet každý záporný prvek a hledat jeho souseda s nejnižší nezápornou hodnotou x. Pokud tento soused leží diagonálně od aktuálního prvku, pak hodnotu aktuálního prvku nastavíme jako $x+\sqrt{2}$ a pokud je vertikálně nebo horizontálně, tak hodnotu nastavíme na $x+1$. Tento algoritmus již vrací na první pohled lepší výsledky, nicméně stále nejsou správné.

Algoritmus, který je použitý pro výpočet mapy vzdáleností, je založen na podobné myšlence jako předchozí algoritmus. Způsob, jakým získává aktuální vzdálenost od nejbližšího černého pixelu, je ale založen na sčítání vektorů a vrací vždy přesné hodnoty. Na začátku vytvoříme dvourozměrné pole vektorů o rozměrech bitmapy podpisu. Následně pro každý černý pixel nastavíme příslušný prvek pole vektorů na nulový vektor a ostatní ponecháme prázdné. Kromě toho si také budeme udržovat seznam souřadnic S_1 všech prvků v poli, do kterých byl v aktuální iteraci zapsán vektor a druhý seznam S_2 obsahující souřadnice prvků, které chceme v aktuální iteraci procházet. Na začátku každé iterace vždy vyprázdníme seznam S_2 a přesuneme do něj prvky ze seznamu S_1. Poté budeme

procházet všechny prvky seznamu S_2 a pro každé sousední políčko p, které je prázdné provedeme následující:

1. Nalezneme souseda p_n, jehož vektor v_n má nejmenší délku ze všech neprázdných sousedů (takový vždy existuje alespoň jeden, protože aktuální prvek byl sousedem vektoru nastaveného v předchozí iteraci).

2. Vypočítáme vektor $v_x = p_n - p$, který určuje směr a vzdálenost k sousedovi p_n.

3. Do políčka p uložíme vektor $v = v_x + v_n$.

4. Souřadnice políčka p uložíme do seznamu S_1.

Opakování těchto iterací provádíme tak dlouho, dokud nejsou všechny prvky pole vektorů zaplněny. Nyní pro zjištění vzdálenosti daného bodu od nejbližšího černého pixelu stačí spočítat velikost vektoru na souřadnicích tohoto bodu. Jako příjemný bonus navíc získáváme i informaci o přesném směru, ve kterém tento pixel leží.

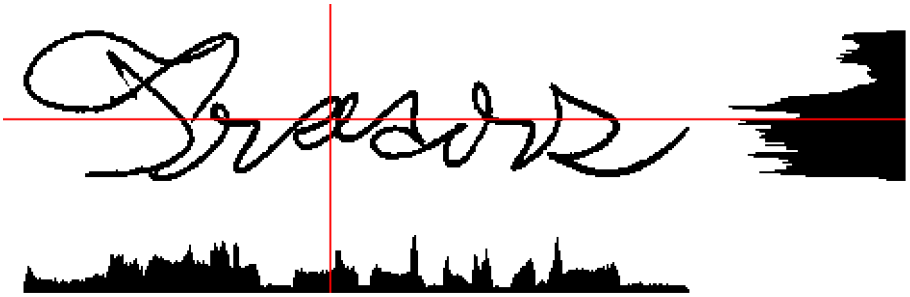
Aby však platilo, že každý vektor v na políčku p s délkou d ukazuje na nejbližší černý pixel, musí pro všechny jeho sousedy v_n a jejich délky d_n platit:

$$|d - d_n| \leq \sqrt{2}, \tag{1}$$

protože pokud vektor v_n na políčku p_n ukazuje na nejbližší černý pixel, pak je tento pixel ve vzdálenosti maximálně $d_n + |p - p_n|$ od políčka p, kde $|p - p_n|$ vyjadřuje vzdálenost mezi sousedními políčky a ta je vždy maximálně $\sqrt{2}$. Tato podmínka však pro tento algoritmus není splněna ve všech případech. Taková situace nastává v momentě, kdy se dva vektory v_1 a v_2 stanou sousedy „přibýváním“ z opačných stran, z nichž například vektor v_1 ukazuje přibližně diagonálním směrem a vektor v_2 přibližně horizontálním nebo vertikálním směrem. I když by vektory vznikly ve stejné iteraci a tedy vzdálenost odpovídajících pixelů od černých pixelů je na počet pixelů stejná, tak reálná délka diagonálně ležícího vektoru bude podstatně delší. Proto může dojít k situaci kdy $d_1 - d_2 > \sqrt{2}$ a z toho vyplývá, že v_1 neukazuje na nejbližší černý pixel. Jednoduchou úpravou totiž získáme $d_1 > d_2 + \sqrt{2}$, což znamená, že pokud vezme vektor v_2 a vektor $v_p = p_2 - p_1$, kde p_1 a p_2 jsou příslušné pixely vektorů v_1 a v_2, pak jejich součtem získáme vektor vycházející z p_1 s menší délkou ukazující na jiný černý pixel.

Tento nedostatek lze odstranit tím, že po naplnění všech prvků pole vektorů budeme v iteracích pokračovat, ale místo prázdných sousedů daného prvku budeme brát sousedy, kteří nesplňují podmínku 1. Algoritmus ukončíme v momentě, kdy po proběhlé iteraci bude seznam S_1 prázdný, tedy nejsou žádné body kterými lze pokračovat v další iteraci.

Protože algoritmus deformace kostry nebude pracovat se souřadnicemi bodů kostry jako s celými čísly, bude při zjišťování vzdáleností docházet k zaokrouhlovací chybě, protože takto konstruovaná mapa poskytuje informace o vzdálenostech pouze pro body s celočíselnými souřadnicemi. K tomu, abychom tuto chybu



Obrázek 14: Horizontální a vertikální souřadnice těžiště podpisu

redukovali na přijatelnou hranici, můžeme algoritmus upravit tak, že každý pixel bitmapy rozdělíme na čtvercovou matici subpixelů $n \times n$ a vzdálenosti budeme počítat přímo pro tyto subpixely. Vzhledem k tomu, že maximální vzdálenost souseda nyní bude $\frac{\sqrt{2}}{n}$, je nutné následovně upravit podmínku 1:

$$|d - d_n| \leq \frac{\sqrt{2}}{n}. \tag{2}$$

3.4 Počáteční umístění podpisů

Základním předpokladem algoritmu ověřování pomocí deformace kostry je vzájemné umístění obou podpisů co nejpřesněji nad sebou. Jedním z možných způsobů jak podpisy vůči sobě umístit je výpočet středu obdélníku těsně obtékajícího podpis, který je určen maximálními a minimálními souřadnicemi černých pixelů na vertikální a horizontální ose. Podpisy tedy umístíme tak, aby tyto středy ležely na sobě. Tento způsob je ale náchylný na protažené krajní linky podpisu, které ovlivňují ohraničující obdélník. Například pokud autor podpisu udělá o něco delší diakritickou čárku než obvykle, pak se to znatelně projeví na výsledném středu obdélníku a tím i na posunutí obou podpisů vůči sobě.

Proto byl pro umístění obou podpisů použitý způsob, ve kterém je středový bod obdélníku obtékajícího podpis nahrazen těžištěm podpisu, které je podstatně méně náchylné na tyto rozdíly. Výpočet souřadnic se provádí na vertikální a horizontální ose zvlášť. Pro horizontální osu z podpisu nejdříve vytvoříme obrazec, který má stejný počet černých pixelů v každém sloupci, ale jsou zarovnány ke spodní hraně bitmapy. Tímto jsme dostali obrazec (obr. č. 14), který bude mít těžiště horizontální osy stejné, jako náš původní podpis, ale bude snazší jej spočítat. Těžiště t na ose X geometrického tvaru definovaného spojitou charakteristickou funkcí $c(x)$ je dáno vztahem:

$$t = \frac{\int x \cdot c(x) dx}{\int c(x) dx} \tag{3}$$

a pro geometrické tvary definované diskrétní charakteristickou funkcí $c(x)$ je dáno vztahem:

$$t = \frac{\sum_i x_i \cdot c(x_i)}{\sum_i c(x_i)}. \tag{4}$$

Vertikální souřadnici těžiště podpisu získáme obdobným způsobem, pokud zaměníme sloupce za řádky.

3.5 Algoritmus deformace kostry

Stěžejním algoritmem celé práce je algoritmus deformace kostry podpisu. To, jakým způsobem bude kostra deformováná na vzorový podpis, zásadně ovlivňuje výsledky celého ověřování a tento algoritmus byl základem pro vypracování této práce.

Kostra podpisu v naší verzi je obdobou tvarově paměťového hada, takže každý uzel kostry zná původní velikosti svých segmentů a také úhlů, které jsou těmito segmenty svírány. Pokud tedy provedeme deformaci libovolným posunutím jednoho uzlu kostry, pak jsme z paměti kostry schopni zjistit míru natažení či stlačení jak segmentů tak úhlů tohoto uzlu. Díky tomu můžeme zjistit souhrnné informace o deformaci celé kostry. Dále jsme schopni pro každý bod kostry zjistit, jak daleko se nachází od nejbližší linky vzorového podpisu, tedy konkrétně od nejbližšího černého pixelu. Díky tomu opět můžeme zjistit souhrnné informace o průměrné vzdálenosti celé kostry od vzorového podpisu. Máme tedy k dispozici tři veličiny, pomocí kterých budeme deformaci řídit.

Na problém deformace lze nahlížet jako na minimalizační problém, kde se snažíme minimalizovat jak vzdálenost uzlů od podpisu, tak i deformaci jednotlivých segmentů a úhlů. Čím více však snižujeme vzdálenost, tím více roste deformace. Algoritmus pracuje v iteracích tak, že v každé iteraci pro každý bod kostry pomocí předem definovaných směrů a vzdáleností hledá nejlepší posunutí tohoto bodu vzhledem k jeho vzdálenosti on linky, deformaci úhlů a deformaci segmentů. Nejlepší posunutí algoritmus hledá včetně nulového, což znamená, že pokud už se bod nachází v optimální pozici a jeho posunutí by znamenalo pouze navýšení některé ze tří posuzovaných složek, pak je bod ponechán na svém místě. Je zřejmé, že důležitou roli hraje způsob, jakým jsou jednotlivé „vstupní veličiny“ určovány a také způsob, jakým posuzujeme celkovou výhodnost (cenu) pozic bodu kostry vzhledem k těmto veličinám.

Určení vzdálenosti od linky

Vzdálenost bodu kostry od nejbližšího černého pixelu vzorového podpisu je určována pomocí *mapy vzdáleností*, která byla popsána v kapitole 3.3. Menším problémem je fakt, že pozice bodu kostry je reprezentována dvojicí reálných čísel, ale mapa vzdáleností má k dispozici pouze informace o vzdálenostech pro souřadnice, které jsou násobky čísla $\frac{1}{n}$, kde n je rozměr čtvercové matice subpixelů, pro které byla mapa vzdáleností spočítána. Pokud však vypočítáme mapu vzdáleností s dostatečně velkou maticí subpixelů, pak můžeme pro potřeby zjištění vzdálenosti bodů kostry souřadnice zaokrouhlovat, protože zaokrouhlovací chyba bude dostatečně malá na to, abychom ji mohli zanedbat.

Výpočet deformace segmentu

Deformace segmentu je vyjádřena jako poměr mezi původní délkou d_0 a aktuální délkou d. Pokud bychom ale stanovili funkci deformace $f_s(d)$ pouze jako poměr těchto dvou čísel, pak vytvoříme nekonzistenci mezi deformací typu stlačení segmentu a deformací typu natažení segmentu, protože pro jeden z těchto dvou typů deformací bude funkce $f_s(d)$ rostoucí a pro druhý bude klesající. Intuitivně bychom chtěli, aby například pro dvojnásobné natažení segmentu vracela funkce $f_s(d)$ stejné hodnoty jako pro dvojnásobné stlačení. Proto je funkce deformace definovaná následovně:

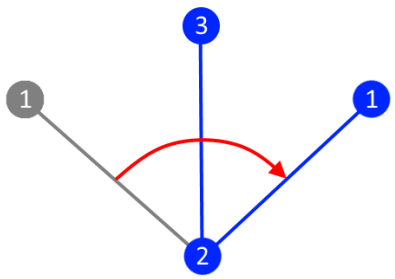
$$f_s(d) = max\left(\frac{d_0}{d}, \frac{d}{d_0}\right). \tag{5}$$

Výpočet deformace úhlu

Deformace úhlu je vyjádřena jako absolutní hodnota rozdílu původního úhlu u_0 a aktuálního úhlu u:

$$f_u(u) = |u_0 - u|. \tag{6}$$

Je však důležité si uvědomit, že nám záleží na znaménku úhlu. Jinak řečeno chceme rozlišovat pořadí bodů daného úhlu vůči původnímu pořadí. Jako příklad si vezměmě dva segmenty svírající počáteční úhel 45° znázorněné na obrázku č. 15. Nyní budeme rotací jednoho segmentu kolem středového bodu úhlu deformovat úhel až na 0° a poté budeme pokračovat stejným směrem ještě dalších 45°. V tuto chvíli budou segmenty opět svírat úhel 45° a výsledná deformace bude 0°, přestože jsme provedli deformaci o celých 90°. Důvodem je právě zanedbání po-



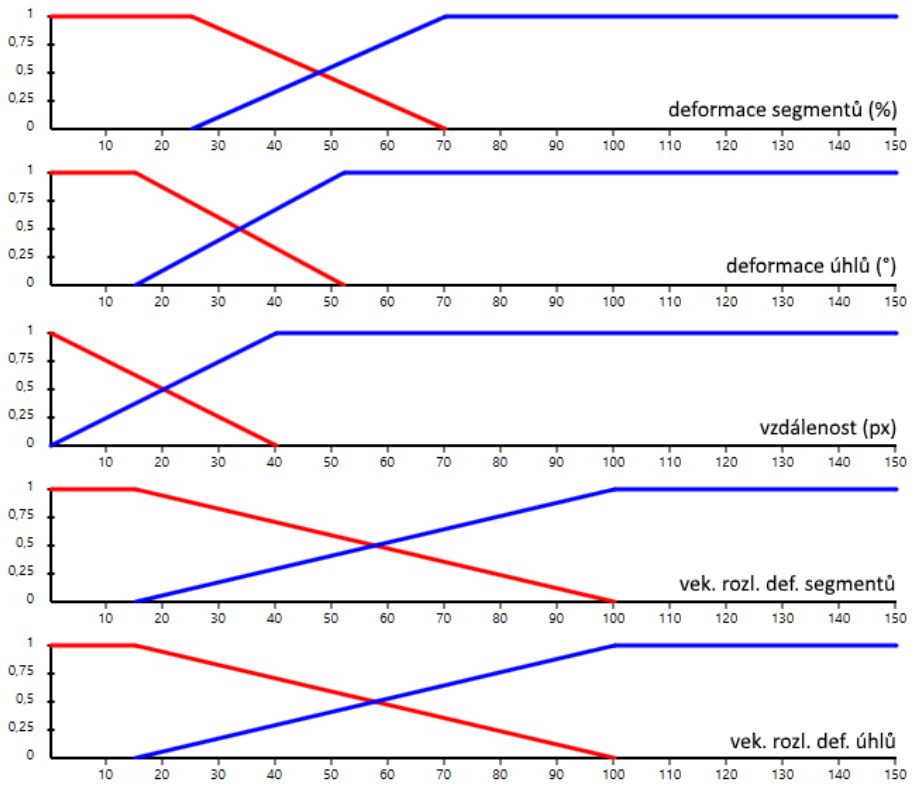
Obrázek 15: Deformace úhlu

řadí, v jakém segmenty úhel svírají. Protože se toto pořadí vlivem deformace změnilo, musíme otočit znaménko úhlu. Nyní tedy dostáváme $|45° - (-45°)| = 90°$.

Vyhodnocení pozice

Způsob, jakým jsou vyhodnocovány výhodnosti pozice a deformace, je založen na fuzzy inferenčním systému, na jehož vstupu jsou hodnoty deformace segmentů, úhlů a vzdálenost. Výstupem je reálné číslo, které shrnuje všechny tři vstupní

složky a naší snahou je ho minimalizovat. Pro tuto část vyhodnocování byl použit Takagi–Sugeno inferenční systém, jehož inferenční pravidla se skládají z fuzzy množiny v antecedentu a ostré hodnoty v konsekventu. Při vyhodnocování jsou pak hodnoty konsekventu modifikovány na základě stupně splnění antecedentu. Pro jednotlivé vstupní veličiny inferenčního systému byly zvoleny fuzzy množiny příslušné k lingvistickým proměnným „malá hodnota“ a „velká hodnota“ tak, jak jsou znázorněny na obrázku č.16. Pro fuzzy množiny označující velkou



Obrázek 16: Fuzzy množiny vstupních veličin vyhodnocování pozice bodu

hodnotu byly vytvořeny inferenční pravidla s těmito množinami v antecedentu a ostrou hodnotou 1 v konsekventu. Obráceně pro fuzzy množiny označující malou hodnotu byly vtvořeny inferenční pravidla s ostrou hodnotou 0. Výstup celého inferenčního systému je vážený součet jednotlivých ostrých hodnot vzniklých vyhodnocením inferenčních pravidel.

Nyní již máme způsob, jakým vypočítávat jednotlivé deformace a vzdálenost a také způsob, jakým tyto hodnoty vyhodnocovat tak, aby bylo možné porovnávat různé posunutí bodů kostry. Při implementaci a následném testování se však ukázalo, že tyto tři vstupní hodnoty nejsou zcela dostačující tak, aby proces postupné deformace probíhal relativně dobře. Deformace jako taková sice probíhala, ale kostra nedostatečně držela svůj tvar a často degenerovala kvůli velikým lokálním deformacím. Proto bylo nutné vložit na vstup vyhodnocovacího systému

ještě další dvě vstupní veličiny, vnášející do algoritmu deformace jakýsi fyzikální pohled na držení tvaru kostry. Hlavní myšlenkou bylo pokusit se pohybem jednoho bodu kostry více ovlivnit jeho sousedy a také rovnoměrně rozložit lokální deformaci do celé kostry.

Vektor rozložení deformace segmentů

Při deformaci kostry dochází k nerovnoměrnému deformování, protože sousední body spojené segmentem se vzájemně nedostatečně ovlivňují. Mějme například tři body kostry a, b, c na jedné přímce a segmenty mezi nimi s_{ab} a s_{bc}. Pokud posuneme bod a směrem k bodu b a stlačíme tak segment s_{ab}, pak tím zcela jistě ovlivníme výslednou hodnotu deformace segmentu pro bod b. Nicméně i když jsme hodnotu deformace zvýšili, tak pozici bodu b to neovlivní. Protože pokud bychom ho posunuli směrem k bodu c a snížili tak deformaci segmentu s_{ab} tak tím současně zvyšujeme deformaci segmentu s_{bc} a algoritmus deformace tedy tímto posunutím nemá co získat.

Z tohoto důvodu byla do algoritmu deformace přidána další vstupní veličina *vektor rozložení deformace segmentů*, který svou délkou vyjadřuje rozdíl mezi deformacemi jednotlivých segmentů daného bodu a také definuje směr pro snížení tohoto rozdílu. Naší snahou je tento vektor minimalizovat, což znamená rovnoměrně rozdělit deformaci mezi ostatní segmenty. Díky tomu se bude v průběhu dalších iterací tato deformace propagovat rovnoměrně do celé kostry a ta tak nebude v takové míře degenerovat a ztrácet informace o svém původním tvaru.

Pro daný bod kostry b a všechny jeho sousední body n_i, se kterými je propojen segmenty s_i, je vektor rozložení deformace segmentů v určen jako

$$v = \sum_{i=1}^{k} v_{s_i} \cdot def(s_i) \cdot sig(s_i), \tag{7}$$

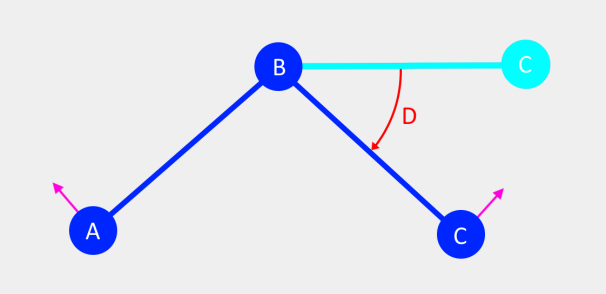
kde $v_{s_i} = \frac{n_i - b}{|n_i - b|}$ je vektor z bodu b do bodu n_i normovaný na délku 1, $def(s_i)$ je funkce vyjadřující míru deformace pro segment s_i a $sig(s_i)$ je funkce vyjadřující znaménko podle typu deformace a je definovaná jako

$$sig(s) = \begin{cases} 1, & \text{pokud } |s| \geq init_s, \\ -1, & \text{jinak,} \end{cases}$$

kde $init_s$ je počáteční velikost segmentu s.

Vektor rozložení deformace úhlů

Stejně jako v předchozím případě, tak i při deformaci úhlů dochází k obdobnému problému nepropagace deformačních sil do sousedních bodů kostry. Zde je ovšem situace o něco složitější, protože posunutím aktuálního bodu kostry nebudeme ovlivňovat přímé sousedy tohoto bodu. K ovlivnění dojde až u sousedů o segment dále, kteří spolu s aktuálním bodem mají společného souseda a jsou tedy body tvořící jeden úhel. Cílem je, aby se kostra chovala co nejvíce pružně a aby se



Obrázek 17: Vektor rozložení deformace úhlu

tím snažila držet svůj původní tvar. Proto každá deformace úhlu bude vytvářet silový potenciál, který ovlivní oba krajní body úhlu.

Mějme body kostry a, b, c a segmenty s_{ab} a s_{bc}, které jsou znázorněny na obrázku č. 17. Posunutím bodu c vznikne na úhlu deformace D. Velikost této deformace stanovíme jako velikost silového potenciálu, který se rovnoměrně rozdělí mezi oba segmenty a tlačí je zpět do původního úhlu. Na krajní body úhlu a a c tak působí silové vektory o velikosti $\frac{D}{2}$, kolmé k daným segmentům. Výsledný vektor rozložení deformace úhlů v je definován jako

$$v = \sum_{i \in I} v_i, \tag{8}$$

kde I je indexová množina sousedů, kteří jsou středovými body nějakého úhlu a v_i je silový vektor i–tého úhlu působící na aktuální bod.

Tento vektor se snažíme opět minimalizovat, což vede k propagaci úhlové deformace do zbytku kostry. Vstupem do inferenčního systému pro vyhodnocení pozice bodu je délka vektoru v.

Přidali jsme tedy dvě nové veličiny na vstup inferenčního mechanismu vyhodnocujícího výhodnost pozice bodů v průběhu deformace. Pro obě veličiny byly definované příslušné fuzzy množiny pro výrazy „velká“ a „malá hodnota“ a pomocí nich také definované nové pravidla inferenčního systému. Tato úprava značně vylepšila chování algoritmu a napomohla k zachování původního tvaru kostry, který vystihuje autora testovaného podpisu. Dokonce v místech, kde se podpisy velmi liší a v předchozí verzi algoritmu docházelo ke kompletní degeneraci, tak nyní omezená pružnost kostry tuto degeneraci nedovolí a deformace se v těchto místech zastaví.

3.6 Vyhodnocení výsledků testu

Celkové vyhodnocení výsledků testování obou podpisů je opět realizováno pomocí inferenčního mechanismu. Jeho pravidla jsou tvořena fuzzy množinami v antecedentu i konsekventu a ostrá hodnota je zvolena jako těžiště sjednocení výsledků

pravidel stejně, jak bylo popsáno v kapitole 2.1.2. Vstupními veličinami inferenčního mechanismu jsou:

- průměrná vzdálenost bodů kostry od podpisové linky,
- průměrná deformace úhlů mezi segmenty kostry,
- průměrná deformace segmentů kostry.

Velikosti vektorů rozložení, které jsme potřebovali navíc v algoritmu deformace, v konečném vyhodnocování nejsou zahrnuty. Tyto veličiny totiž nenesou žádnou důležitou informaci o výsledné deformaci kostry, která by již nebyla obsažena v předešlých třech vstupních veličinách. Z vyhodnocení celkových výsledků je tedy můžeme vypustit. Výstupní veličinou je hodnota od 0 do 100 vyjadřující procentuální „pravděpodobnost“, že autorem obou podpisů byl stejný člověk. Nejedná se však o pravděpodobnost v pravém slova smyslu, ale pouze o známku odvozenou inferenčním systémem. Pro vstupní veličiny deformace segmentů a úhlů byly definovány tři fuzzy množiny *malá hodnota*, *střední hodnota* a *velká hodnota*. Pro veličinu vzdálenosti byla navíc definována fuzzy množina *extrémní hodnota*. Pro výstupní veličinu pravděpodobnosti byly zvoleny čtyři množiny vyjadřující *nulovou, malou, střední* a *velkou* pravděpodobnost. Pomocí těchto fuzzy množin jsou v systému definována inferenční pravidla, která jsou uvedena v tabulce č. 1.

Tabulka 1: Inferenční pravidla

vzdálenost	*není*	extrémní	$\wedge$				
vzdálenost	*není*	velká	$\wedge$				
def. segmentů	*je*	malá		$\Longrightarrow$	pravděpodobnost	*je*	velká
def. segmentů	*je*	střední		$\Longrightarrow$	pravděpodobnost	*je*	střední
def. segmentů	*je*	velká		$\Longrightarrow$	pravděpodobnost	*je*	malá
vzdálenost	*není*	extrémní	$\wedge$				
vzdálenost	*není*	velká	$\wedge$				
def. úhlů	*je*	malá		$\Longrightarrow$	pravděpodobnost	*je*	velká
def. úhlů	*je*	střední		$\Longrightarrow$	pravděpodobnost	*je*	střední
vzdálenost	*není*	malá	$\wedge$				
def. úhlů	*je*	velká		$\Longrightarrow$	pravděpodobnost	*je*	velká
vzdálenost	*je*	malá		$\Longrightarrow$	pravděpodobnost	*je*	velká
vzdálenost	*je*	velká		$\Longrightarrow$	pravděpodobnost	*je*	malá
vzdálenost	*je*	extrémní		$\Longrightarrow$	pravděpodobnost	*je*	nulová

Jak lze vidět, některá pravidla mají antecedent složený z více než jedné podmínky. Důvodem je fakt, že nejvíce směrodatná vstupní veličina je pro nás vzdálenost. Je tedy nutné ostatním veličinám přidat určitou závislost na hodnotu vzdálenosti. Konkrétně chceme, aby nebyla přiřazována vysoká hodnota pravděpodobnosti v situacích, kdy deformace segmentů nebo úhlů vychází malá, ale celková hodnota vzdálenosti je vysoká. Podmínku „vzdálenost *není* velká“ snadno vyjádříme pomocí komplementu fuzzy množiny pro velkou vzdálenost.

4 Implementace

Při implementaci této práce byl, vzhledem k náročnosti problematiky, kladen velký důraz na architekturu celé aplikace, která by umožňovala rychlé a pohodlné změny jednotlivých algoritmů, datových struktur nebo například grafického uživatelského rozhraní. Taková architektura programu na jednu stranu usnadňovala vývoj a dávala možnost vytvářet více implementací řešení jednoho problému, které poté bylo možné vzájemně porovnávat. Na stranu druhou dává možnost navázání dalších autorů na tuto práci a podrobně se tak věnovat jednotlivým podproblémům, jako je například algoritmus konstrukce kostry podpisu.

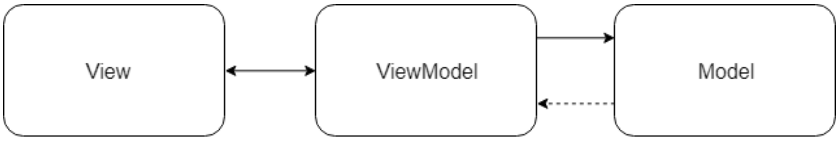
Celé řešení aplikace je složeno ze dvou projektů. Projekt **SignatureTesting** obsahuje třídy implementující hlavní logiku algoritmů, týkajících se ověřování podpisů, datové struktury a kompletní uživatelské rozhraní aplikace. Projekt **Fuzzy** obsahuje třídy implementující fuzzy logiku a fuzzy inferenční mechanismy, které využívají algoritmy ověřování.

Aplikace je implementovaná v jazyku C# postaveném na frameworku *Microsoft .NET Framework 4.5.2* pro platformu *Windows*. Jádrem frameworku je prostředí potřebné pro běh aplikací nabízející spouštěcí rozhraní a široké spektrum knihoven. Není předepsaný konkrétní programovací jazyk, protože aplikace se vždy překládá do mezijazyka Common Intermediate Language. Pro vývoj .NET aplikací je primárně určeno vývojové prostředí *Microsoft Visual Studio* dostupné ve verzích Community, Enterprise a Professional s různými licenčními podmínkami. Pro tuto práci bylo použito Microsoft Visual Stidio Community 2015.

Samotný jazyk C# je vysokoúrovňový, objektově orientovaný programovací jazyk, vyvinutý současně s platformou .NET Framework. Vychází z jazyků C++ a Java. Lze jej využít pro tvorbu formulářových aplikací ve Windows, databázových programů, webových aplikací a služeb, softwaru pro mobilní zařízení a podobně.

4.1 Architektura programu

Celá vnitřní stavba aplikace je navržena podle vzoru *Mode–View–ViewModel (MVVM)*. Jak je patrné už z názvu tohoto vzoru, rozděluje aplikaci na tři části a to podle jejich úlohy v rámci aplikace. Část *View* je pouze prezentační a obstarává výhradně grafické uživatelské rozhraní. Neimplementuje žádnou logiku aplikace a je zodpovědná pouze za zobrazení aplikace na displej. *Model* má právě opačnou úlohu. Obstarává vždy veškerou logiku aplikace včetně práce s databází či se souborovým systémem. Třetí část *ViewModel* je prostředníkem právě mezi View a Modelem a jeho úloha je adaptovat Model pro potřeby View. Je zodpovědný za stav View, avšak bez potřeby zjišťovat stav konkrétních prvků View. Vystavuje veřejné vlastnosti, které View čte a notifikuje o změnách těchto vlastností, aby na ně View mohlo reagovat.



View aplikace

Hlavní třídou části View je MainWindow. Tato třída samotná nedefinuje žádné konkrétní ovládací prvky grafického rozhraní, ale pouze základní styl okna, ikony a podobně. Kromě toho má však zodpovědnost za změnu obsahu okna. Je notifikována třídou MainWindowViewModel o změnách stavu aplikace a podle těchto změn nastavuje zobrazovaný obsah aplikace. Má tedy povědomí o všech třídách, které mohou tento obsah tvořit a má přístup k jejich konstruktorům, aby je mohla dynamicky vytvářet.

ViewModel aplikace

Každý ViewModel v aplikaci vždy dědí z třídy PropertyChangeNotificator. Tato třída implementuje funkcionalitu notifikací definovaných systémovým rozhraním INotifyPropertyChanged. Její implementace dovoluje notifikovat View o změnách i z jiného, než z GUI vlákna aplikace a tím výrazně zjednodušuje práci s vlákny v rámci ViewModelů a zefektivňuje využití vláken v aplikaci.

Hlavní třídou části ViewModel je třída MainWindowViewModel. Její úlohou je přijímat od všech ViewModelů jednotlivých obsahů okna notifikace o snahu změny obsahu okna, zpracovávat je a dále o těchto změnách notifikovat MainWindow s informací o jakou změnu obsahu se jedná.

K tomu, aby MainWindowViewModel byl schopný přijímat notifikace od jednotlivých ViewModelů, musí každý z nich implementovat rozhraní IWindowContentViewModel. To definuje název a tvar notifikace o snahu změny obsahu okna, k jejímuž odběru se může MainWindowViewModel přihlásit. Toto rozhraní však přímo implementuje pouze třída ContentWindowViewModelBase, ze které dále všechny ViewModely obsahů okna dědí. Navíc ContentWindowViewModelBase poskytuje svým potomkům implementaci metody SwitchContent, která celý proces notifikace obstarává a ti jsou tak odstínění od problémů vyšší vrstvy. Samotná třída ContentWindowViewModelBase dědí ze zmiňované třídy PropertyChangeNotificator.

Datový model a funkcionalita

Architektura MVVM zajišťuje aplikaci nejenom přehledné rozdělení uživatelského rozhraní a logiky aplikace, ale zároveň dává možnost snadno měnit jednotlivé části bez ovlivnění funkce ostatních částí. Ve stejném duchu je také navržena architektura modelové části aplikace.

Aby byla zajištěna vysoká modulárnost celé aplikace, jsou třídy tvořící tuto část logicky rozděleny do dvou skupin. První skupinou jsou třídy datové, jejichž

úlohou je pouze uchovávat data svým vnitřním stavem a poskytovat pouze základní nástroje (veřejné metody) pro přístup a práci s těmito daty. Samotné datové třídy většinou neprovádějí žádné změny na svých datech ani žádné složitější výpočty. Druhou skupinou jsou třídy algoritmické, které mají právě opačnou úlohu než třídy datové. Jejich úkolem je vždy pouze poskytovat funkcionalitu nad třídami datovými. Algoritmické třídy reprezentují vždy konkrétní algoritmus řešící komplexnější problém a samotné si buďto neudržují žádný vnitřní stav nebo pouze dočasný v rámci jednoho výpočtu. Datové třídy jsou tedy modifikovány právě třídami algoritmickými, které k modifikaci využívají pouze nástroje poskytované datovými třídami.

Tato architektura dává možnost měnit jednotlivé algoritmy bez toho, abychom museli měnit nějakým zásadním způsobem zbytek kódu. Stejně tak můžeme pohodlně měnit interní reprezentaci dat bez toho, aby bylo nutné upravovat chod jednotlivých algoritmů.

Algoritmy definované rozhraním

Každá třída reprezentující konkrétní algoritmus vždy implementuje dané rozhraní definující vstupy a výstupy tohoto algoritmu. V rámci aplikace se k objektům všech algoritmů přistupuje výhradně pomocí příslušného rozhraní. Tím je zajištěna stoprocentní kompatibilita jakékoliv třídy algoritmu implementující příslušné rozhraní. Kromě snadné záměny výchozích algoritmů také může aplikace za běhu dynamicky měnit používané algoritmy pro dané problémy.

```
1  public interface ISnakeBuildAlgorithm
2  {
3     Snake Run(Px[,] signatureArray);
4  }
5
6  public interface ISnakeAdjustmentAlgorithm
7  {
8     Task Run(Snake snake, DistanceMap distanceMap, Action<double>
          progressSetter);
9
10    void BeforeRun(object settings);
11 }
12
13 public interface ISignatureCenterAlgorithm
14 {
15    Point Run(Px[,] signatureArray);
16 }
```

Zdrojový kód 1: Ukázka rozhraní algoritmů

4.2 Implementace kostry podpisu

Kostra podpisu je v aplikaci implementována jako datová třída Snake, která obsahuje kolekci jednotlivých bodů kostry implementovaných samostatnou třídou. Třída Snake jako datová třída neimplementuje žádnou funkcionalitu měnící její data, ale poskytuje metody pro výpočet průměrných deformací a vzdálenosti každého uzlu kostry. Tyto hodnoty jsou následně vstupem pro vyhodnocovací algoritmus.

```
1    public class Snake
2    {
3       public Snake(List<SnakePoint> points){SnakePoints = points;}
4
5       public List<SnakePoint> SnakePoints { get; }
6       public void RememberShape() {...}
7       public double AverageSegmentDeformation { get { ... } }
8
9       public double AverageAngleDeformation
10      {
11         get
12         {
13            var count = 0; double sum = 0;
14            foreach (var snakePoint in SnakePoints)
15               foreach (var angle in snakePoint.Angles)
16               {
17                  sum += angle.GetDeformation();
18                  count++;
19               }
20            return sum / count;
21         }
22      }
23
24      public double AverageDistance(DistanceMap distanceMap)
25      {
26         double sum = 0;
27         foreach (var snakePoint in SnakePoints)
28            sum += distanceMap.GetValue(snakePoint.X, snakePoint.Y);
29         return sum / SnakePoints.Count;
30      }
31   }
```

Zdrojový kód 2: Implementace třídy Snake

Implementace jednotlivých bodů kostry je realizována třídou SnakePoint. Tato třída je opět datovou a je jádrem implementace celé kostry. Třída Snake je zde pouze obalující třídou odstiňující zbytek aplikace od práce s kolekcí objektů třídy SnakePoint. Každá instance třídy SnakePoint zná své přesné souřadnice a obsahuje kolekci svých sousedů, s nimiž je propojena segmentem kostry. Pro vytvoření „tvarové paměti“ kostry jsou zde třídy SegmentMemory a AngleMemory. Ty slouží

jako paměť velikosti jednotlivých segmentů a úhlů. Také poskytují metody pro výpočet deformace, která je vždy stanovena z rozdílu mezi aktuálním stavem a stavem při vytvoření konkrétních instancí SegmentMemory a AngleMemory. Instance třídy SnakePoint pak obsahuje kolekci instancí SegmentMemory pro každý její segment a kolekci instancí AngleMemory pro každý úhel svírající dvěma segmenty dané instance třídy SnakePoint.

```
1       public class SegmentMemory
2       {
3          public SegmentMemory(SnakePoint point, SnakePoint neighb)
4          {
5             Point = point; Neighbour = neighb;
6             InitialSize = Distance(point, neighb);
7          }
8          public SnakePoint Point { get; }
9          public SnakePoint Neighbour { get; }
10         public double InitialSize { get; }
11         public int DeformSig => Distance(Point, Neighbour) -
               InitialSize >= 0 ? 1 : -1;
12
13         public double GetDeformation()
14         {
15            double actualSize = Distance(Point, Neighbour);
16            var max = Math.Max(actualSize, InitialSize);
17            var min = Math.Min(actualSize, InitialSize);
18            var result = ((max / min) - 1) * 100;
19            if (double.IsNaN(result || double.IsInfinity(result ))
20               return 0;
21            return result;
22         }
23
24         private double Distance(SnakePoint p1, SnakePoint p2)
25         {
26            var dX = p1.X - p2.X; var dY = p1.Y - p2.Y;
27            return Math.Sqrt(dX * dX + dY * dY);
28         }
29      }
```

Zdrojový kód 3: Implementace třídy SegmentMemory

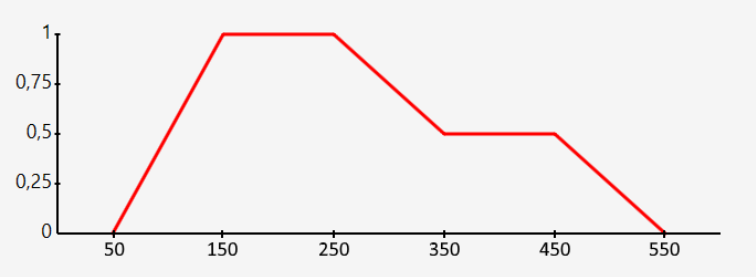
4.3 Implementace fuzzy množin

Pro potřeby této práce bylo nutné vytvořit sadu tříd, jež by reprezentovaly fuzzy množiny a umožňovaly by základní operace, které jsou nutné pro implementaci fuzzy inferenčních mechanismů. Implementace této funkcionality byla vytvořena v odděleném projektu **Fuzzy** tak, aby byla po kompilaci nezávislou knihovnou a bylo ji možné dále využívat i v jiných aplikacích.

Implementaci fuzzy množiny realizuje třída FuzzySet. Ta je interně reprezentována lomenou čárou, která popisuje průběh charakteristické funkce fuzzy množiny. Vstupem pro konstruktor třídy FuzzySet je seznam bodů definujících koncové body jednotlivých úseček lomené čáry. Tyto body jsou instancemi třídy FuzzySetPoint. Na ose X mohou nabývat libovolných souřadnic a na ose Y mohou nabývat hodnot z intervalu $\langle 0, 1 \rangle$. Osa X tedy představuje prvky univerza a osa Y vyjadřuje stupeň příslušnosti těchto prvků k dané fuzzy množině. Příklad vytvoření instance třídy FuzzySet ukazuje zdrojový kód č. 4.

```
1  FuzzySet f1 = new FuzzySet(new List<FuzzySetPoint>()
2  {
3          new FuzzySetPoint(50, 0),
4          new FuzzySetPoint(150, 1),
5          new FuzzySetPoint(250, 1),
6          new FuzzySetPoint(350, 0.5),
7          new FuzzySetPoint(450, 0.5),
8          new FuzzySetPoint(550, 0),
9  });
```

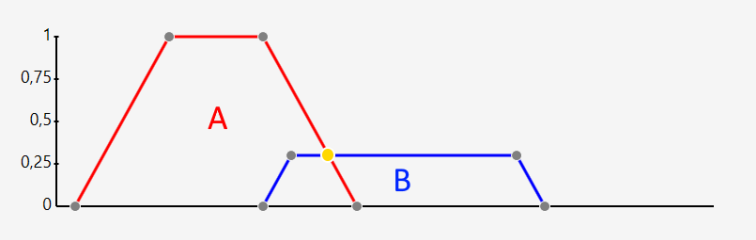
Zdrojový kód 4: Vytvoření instance třídy FuzzySet



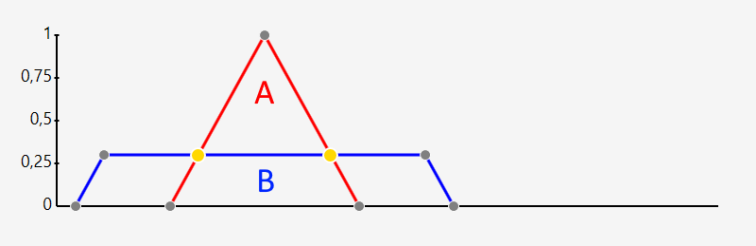
Obrázek 18: Fuzzy množina vytvořená zdrojovým kódem č. 4.

Pro zjištění stupně příslušnosti konkrétního prvku (čísla), implementuje FuzzySet metodu DegreeOfMembership(double x). Ta pro číslo x nalezne příslušnou úsečku lomené čáry, mezi jejíž koncové body x spadá a z ní vypočítá stupeň příslušnosti pro x. Pokud se však x nachází před prvním nebo za posledním bodem fuzzy množiny, pak je stupeň roven stupni příslušnosti bližšímu z těchto dvou bodů. Jsme tedy schopni zjistit přesný stupeň příslušnosti, pro kterýkoliv prvek univerza a tento způsob reprezentace je tak velmi efektivní vzhledem k množství informací, které uchovává.

Situace se komplikuje v implementaci množinových operací sjednocení a průniku. Jejich výpočet pro dvě fuzzy množiny reprezentované pomocí bodů lomené čáry není zcela triviální. Při těchto operacích jsou totiž některé původní body obou množin odstraněny a některé nové body naopak vznikají. Vezměme například dvě fuzzy množiny znázorněné na obrázku č. 19. Pokud provedeme



Obrázek 19: Body implementace fuzzy množiny



Obrázek 20: Problém hledání průsečíků hran

jejich sjednocení, pak ve výsledné množině nebude obsažen poslední bod množiny A a první dva body množiny B. Kromě toho nám ale vzniká nový, který bude součástí výsledného sjednocení a který vznikl křížením hran obou množin. Podobně, pokud provedeme operaci průniku, bude výsledná množina obsahovat první dva body množiny B, poslední bod množiny A a opět nový bod vzniklý jako průsečík hran obou množin. Problém toho, který bod bude ve výsledku obsažen a který bude odstraněn, je řešený porovnáváním stupně příslušnosti tohoto bodu se stupňem druhé množiny pro stejnou souřadnici x. Pro sjednocení bude bod odstraněn, pokud je jeho stupeň nižší a pro průnik bude odstraněn, pokud je jeho stupeň vyšší než stupeň druhé množiny.

Problém zjištění, zda existuje a kde se nachází průsečík, kterým vznikne nový bod, je podstatně těžší, pokud nechceme tuto úlohu řešit hrubou silou. Základní myšlenkou efektivní implementace je sledování, jak se mění vztahy stupňů příslušnosti obou množin v jednotlivých bodech. Například na obrázku č. 19 je stupěň množiny A v prvním, druhém a třetím bodě vyšší než stupeň množiny B. Pro čtvrtý bod už ale tento vztah neplatí a tato změna indikuje průsečík. Tento přístup má však při reálné implementaci několik úskalí. Jedním z nich je například fakt, že je nutné takto porovnávat body obou množin vzhledem k té druhé. Změna vztahu stupňů příslušnosti množin se nemusí nutně projevit pro body obou fuzzy množin. Tento případ je vidět na obrázku č. 20, kde množina B má pro všechny její body vyšší stupeň, než množina A a to i přes to, že došlo ke dvěma protnutím hran.

Implementační řešení operace sjednocení a průniku obsahuje třída FuzzySet v metodách Union(FuzzySet f) a Intersection(FuzzySet f). Kromě těchto binárních operací s fuzzy množinami obsahuje třída FuzzySet také implementaci unární operace doplňku, kterou realizuje metoda Complement(double start, double end).

Parametry této metody start a end specifikují počátek a konec univerza, na kterém má být doplněk původní množiny proveden. Poslední z důležitých metod třídy FuzzySet je metoda Cut(double degree). Ta očekává jako vstupní parametr číslo z intervalu $\langle 0, 1 \rangle$ vyjadřující stupeň, na který má být původní fuzzy množina „oříznuta“. Příklady použití těchto metod jsou uvedeny ve zdrojovém kódu č. 5.

```
1  FuzzySet f1 = new FuzzySet(new List<FuzzySetPoint>()
2  {
3          new FuzzySetPoint(50, 0),
4          new FuzzySetPoint(150, 1),
5          new FuzzySetPoint(250, 0),
6  });
7
8  FuzzySet f2 = new FuzzySet(new List<FuzzySetPoint>()
9  {
10         new FuzzySetPoint(150, 0),
11         new FuzzySetPoint(250, 1),
12         new FuzzySetPoint(350, 0),
13 });
14
15 FuzzySet unionResult = f1.Union(f2);
16 FuzzySet intersectResult = f1.Intersection(f2);
17 FuzzySet complementResult = f1.Complement(0,400);
18 FuzzySet cutResult = f1.Cut(0.45);
```

Zdrojový kód 5: Použití metod třídy FuzzySet

Implementace inferenčního mechanismu

Mechanismus inferenčního vyhodnocování implementuje třída FuzzyInferention. Její konstruktor očekává seznam inferenčních pravidel, kterými bude proces vyhodnocování řízen. Pravidla jsou uložena jako vnitřní stav objektu a jsou využívána pro každé vyhodnocování. Není tedy nutné vytvářet nebo předávat pravidla znovu pro každé vyhodnocení.

Samotné vyhodnocení pravidel provádí metoda Eval(Tuple<double,string>[] values). Vstupem této metody je pole dvojic, kde první prvek z dvojice je desetinné číslo reprezentující aktuální hodnotu vstupní veličiny a druhý prvek je řetězec s názvem veličiny. Díky tomu, že vstupem jsou dvojice hodnot a jmen, tak nezáleží na pořadí veličin v poli, protože je možné každou hodnotu identifikovat právě podle jména veličiny. Metoda Eval prochází přes všechny inferenční pravidla a pro každé z nich vybere správné vstupní hodnoty, které jsou definovány antecedentem tohoto pravidla a provede jeho vyhodnocení. Výsledek vyhodnocení pravidla vždy sjednotí s výsledky vyhodnocení předchozích pravidel. Návratovou hodnotou metody je desetinné číslo, vzniklé jako těžiště výsledného sjednocení všech výsledků pravidel.

```
1  public double Eval(Tuple<double,string>[] values)
2  {
3      var resultFuzzySet = FuzzySet.EmptySet;
4      foreach (var rule in Rules)
5      {
6          var inputValues = new double[rule.Antecedent.Length];
7          foreach (var tuple in values)
8          {
9              var index = Array.IndexOf(rule.AntecedentNames, tuple.
                   Item2);
10             if (index >= 0)
11                inputValues[index] = tuple.Item1;
12         }
13         var currentSet = rule.Eval(inputValues);
14         resultFuzzySet = resultFuzzySet.Union(currentSet);
15     }
16     return resultFuzzySet.CenterOfGravity();
17 }
```

Zdrojový kód 6: Implementace metody Eval ve třídě FuzzyInferention

Pravidla inferenčního mechanismu jsou implementována jako samostatná třída FuzzyRule, která obsahuje dva přetížené konstruktory se dvěma parametry antecedent a consequent. První konstruktor má jako parametr antecedent dvojici ⟨FuzzySet, string⟩ definující jednu fuzzy množinu antecedentu a jméno vstupní veličiny příslušnou k této fuzzy množině. Druhý konstruktor bere jako parametr antecedent libovolnou kolekci těchto dvojic implementující rozhraní *IEnumerable*. Antecedent pravidel tedy dovoluje obsahovat více fuzzy množin různých vstupních veličin. Druhý parametr consequent je v obou případech fuzzy množina příslušná výstupní veličině pravidla. Konstruktory vstupní parametry zpracují a vytvoří veřejné pole fuzzy množin Antecedent a k němu odpovídající veřejné pole jmen veličin AntecedentNames, které bude definovat uspořádání vstupních hodnot při vyhodnocování.

K vyhodnocení pravidla obsahuje třída FuzzyRule metodu Eval(double[ ] inputValues). Parametrem jsou aktuální hodnoty vstupních veličin, které jsou obsaženy v antecedentu daného pravidla a to v pořadí, které definuje veřejné pole AntecedentNames. Výpočet metody Eval probíhá tak, že pro každou vstupní hodnotu zjistíme stupeň příslušnosti k fuzzy množině stejné veličiny. Následně nalezneme nejmenší z těchto stupňů a pomocí metody Cut „ořízneme“ fuzzy množinu konsekventu na tento stupeň. Takto vzniklá množina je výsledkem vyhodnocení inferenčního pravidla a je vrácena jako návratová hodnota metody Eval.

4.4 Použité technologie

Aplikace využívá několik knihoven určených pro tvorbu grafického uživatelského rozhraní. Všechny použité knihovny jsou buďto součástí frameworku .NET, nebo

```
1  public FuzzySet Eval(double[] inputValues)
2  {
3     var cutLevel = double.PositiveInfinity;
4     for (int i = 0; i < Antecedent.Length; i++)
5     {
6        var degree = Antecedent[i].DegreeOfMembership(inputValues[i]);
7        if (degree < cutLevel)
8            cutLevel = degree;
9     }
10    return Consequent.Cut(cutLevel);
11 }
```

Zdrojový kód 7: Implementace metody Eval ve třídě FuzzyRule

je lze získat pomocí balíčkovacího systému NuGet poskytovaného vývojovým prostředím Visual Studio.

Windows Presentation Foundation

Windows Presentation Foundation (WPF) je knihovna pro tvorbu grafického rozhraní obsahující velké množství ovládacích a zobrazovacích komponent GUI. Je součástí .NET frameworku od verze 3.0 a je nástupcem zastaralé knihovny Windows Forms. WPF umožňuje využívat značkovací jazyk *XAML*, pomocí něhož lze oddělit vzhled aplikace od jeho funkčnosti. Aplikace pak mohou být velmi pohodlně a přehledně implementovány pomocí architektury MVVM.

Mahapps

Mahapps je volně dostupná open source knihovna designových stylů pro aplikace tvořené ve WPF. Obsahuje definice stylů jak pro jednotlivé prvky uživatelského rozhraní jako jsou *check box, combo box, group box* a podobně, tak i styly pro celé okna aplikace. Mimo základní prvky grafického rozhraní poskytované knihovnou WPF, definuje nové prvky jako je například *hamburger menu* nebo *MetroHeader*. Dále také obsahuje velkou sadu ikon definovaných vektorovou grafikou pro nejrůznější využití.

Fody

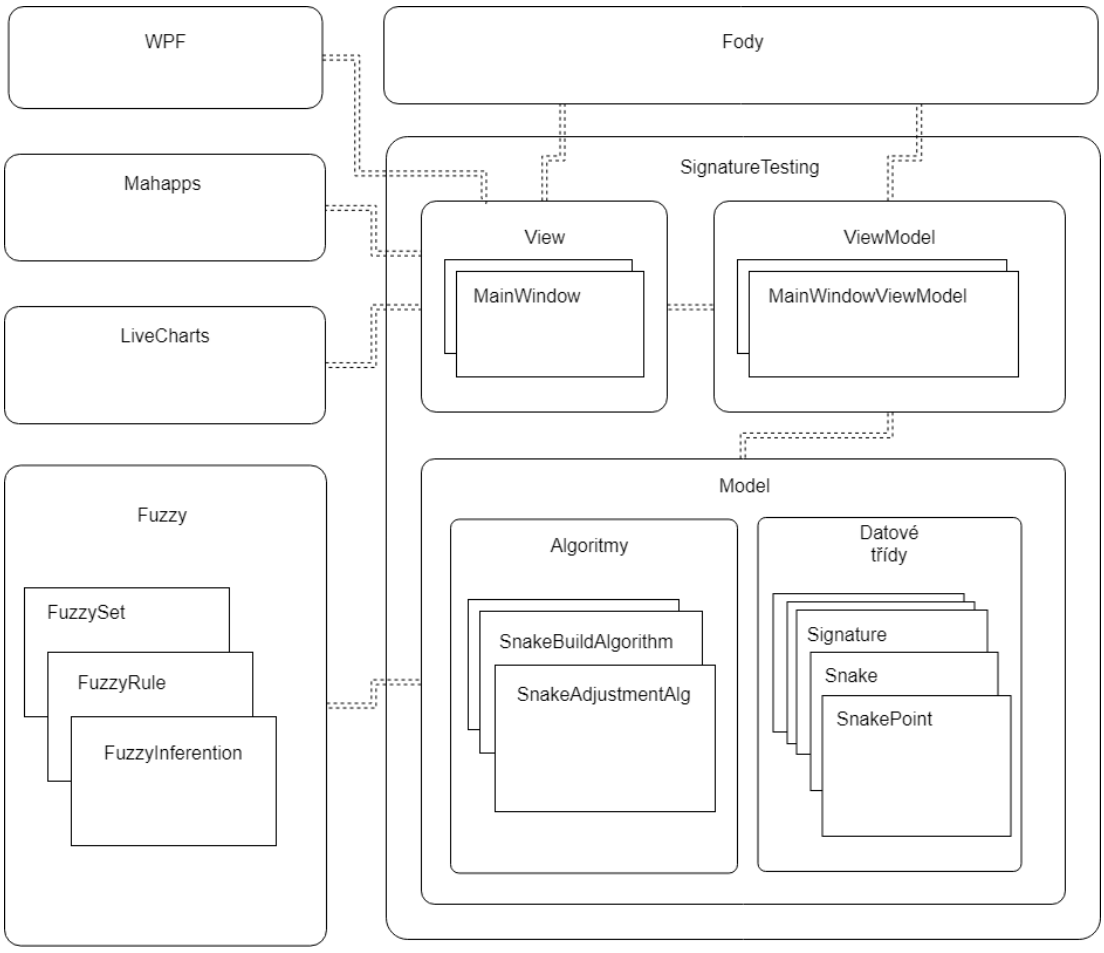
Knihovna WPF používá vlastní notifikační systém pro oznámení změn dat, které jsou zobrazovány pomocí uživatelského rozhraní. Aby tato notifikace fungovala, je nutné, aby třídy dat, které GUI zobrazuje, implementovaly rozhraní *INotifyPropertyChanged* a při každé změně je nutné volat metodu *OnPropertyChanged* s podrobnostmi o této změně. Knihovna Fody umí kromě jiného tuto běžnou a rutinní implementaci notifikace změn doplnit sama a oprošťuje vývojáře od povinnosti psaní stále se opakujícího kódu, který navíc bývá častým zdrojem chyb.

LiveCharts

Pro grafické zobrazování číselných dat byla použita knihovna LiveCharts poskytující široké spektrum grafů a diagramů. Knihovna nabízí možnost nastavení animací změn, legendy, tooltipy, barevné rozvržení atd.

4.5 Logické schéma aplikace

Následující schéma popisuje kompletní architekturu a logické uspořádání aplikace v kontextu MVVM přístupu a dělení tříd na datové a algoritmické. Znázorňuje také všechny knihovny, které využívají jednotlivé logické části aplikace a vzájemné závislosti na těchto knihovnách a ostatních částech programu.



Obrázek 21: Logické schéma aplikace

5 Uživatelská příručka

Tato kapitola popisuje způsob, jakým používat aplikaci ověřování podpisu a její jednotlivé prvky grafického uživatelského rozhraní. Program uživatele provádí

jednotlivými kroky ověření pomocí přecházení mezi vzájemně navazujícími obrazovkami jednotlivých kroků.

5.1 Systémové požadavky

Aplikace byla vyvinuta pro platformu Windows a běží na frameworku .NET. Pro bezproblémové spuštění je nutné mít nainstalovaný .NET Framework ve verzi 4.5.2 nebo novější. Vývoj a testování probíhalo na 64–bitovém operačním systému Windows 10 Home s nainstalovaným .NET Framewok 4.6.1 (doporučeno). Grafické uživatelské rozhraní vyžaduje minimální rozlišení 800x600 a bylo optimalizováno pro rozlišení Full HD 1920x1080 (doporučeno).

5.2 Instalace

Součástí této práce je přiložené CD/DVD, které obsahuje adresář s názvem `bin`. V tomto adresáři se nachází instalační soubor `setup.exe`. Spuštěním tohoto souboru spustíte kompletní instalaci aplikace. Podrobné informace pro bezproblémovou instalaci aplikace jsou obsaženy v souboru `readme.txt`, který je součástí obsahu přiloženého CD/DVD.

5.3 Spuštění programu a úvodní obrazovka

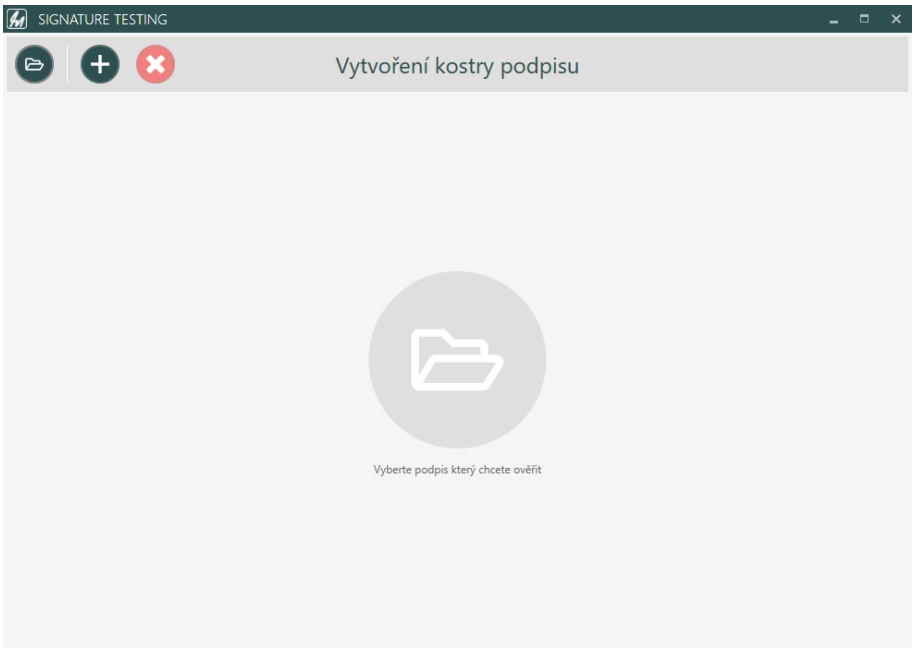
Po instalaci programu není třeba vykonávat žádné další nastavení. Program spustíte dvojklikem na ikonu aplikace. Následně se otevře program s úvodní obrazovkou. Pro zahájení testu podpisů klikněte na tlačítko „ZAČÍT TEST“ uprostřed obrazovky.

Obrázek 22: Úvodní obrazovka

5.4 Obrazovka editoru kostry podpisu

Po stisku tlačítka „ZAČÍT TEST“ přejdete na obrazovku editoru kostry podpisu. Aplikace totiž kromě výchozího algoritmu pro konstrukci kostry podpisu také nabízí editor této kostry za účelem zvýšení kvality výsledku ověření. Kostru je zde možné libovolně deformovat, přidávat a odebírat body nebo přidávat a odebírat hrany (segmenty) kostry.

Uprostřed obrazovky se nachází tlačítko s ikonou složky pro výběr podpisu, jehož pravost chce uživatel otestovat. Po stisku tohoto tlačítka se objeví standartní dialog pro výběr souboru s podpisem. Aplikace očekává bitmapový obrázek ve formátu PNG nebo JPG o rozměrech 500×300 pixelů.



Obrázek 23: Obrazovka editoru kostry podpisu

Výběr testovaného podpisu

Po zvolení souboru s testovaným podpisem, bude soubor zpracován a načten aplikací. Následně se podpis zobrazí na obrazovce včetně předběžné kostry podpisu vyznačené modrou barvou. Pokud chce uživatel načíst jiný podpisový soubor, může tak učinit stisknutím tlačítka s ikonou složky v levém horním rohu. Opět se zobrazí standartní dialog pro výběr nového souboru.

Editace kostry

Ovládání editoru se provádí pomocí kurzoru myši. Přesunutím kurzoru nad plochu podpisového obrázku a současným otáčením kolečka myši lze libovolně měnit přiblížení podpisu pro lepší práci v editoru. Přiblížení je vždy měněno vzhledem

Obrázek 24: Obrazovka editoru kostry podpisu - načtený podpis

k bodu, nad kterým se právě nachází kurzor myši. Posunutí celého podpisu po pracovní ploše se provádí stiskem pravého tlačítka myši nad plochou podpisového obrázku a následným tahem libovolným směrem.

Pohyb bodu: Stiskem levého tlačítka myši nad kterýmkoliv bodem kostry podpisu a následným tažením kurzoru libovolným směrem lze měnit pozici bodu kostry. Spolu s bodem se bude také měnit poloha a délka hran vedoucí do tohoto bodu.

Označení bodu: Označení konkrétního bodu lze provést pomocí dvojkliku levým tlačítkem myši na tento bod. Označení se projeví tak, že bod změní barvu z modré na zelenou. Odznačení tohoto bodu provedete opětovným dvojklikem na tento bod.

Přidání hrany: Pro přidání hrany (segmentu) mezi dvěma body kostry nejprve označte dvojklikem jeden z těchto bodů. Poté proveďte dvojklik na druhý z bodů. Hrana mezi body bude automaticky vytvořena a výběr prvního bodu bude zrušen.

Odstranění hrany: Odstranění hrany (segmentu) mezi dvěma body kostry probíhá analogicky jako přidání. Nejprve označte dvojklikem jeden z těchto bodů. Poté proveďte dvojklik na druhý z bodů. Hrana mezi body bude odstraněna a výběr prvního bodu bude zrušen.

Přidání bodu: Kostru podpisu lze také modifikovat přidáváním bodů. Pro přidání stiskněte zelené tlačítko s ikonou „+“ v horní liště aplikace. Nyní bude nový bod kostry přichycen na kurzoru myši. Pro umístění tohoto bodu na konkrétní místo stiskněte levé tlačítko myši. Tento bod lze opět posouvat a přidávat k němu nové hrany jako u předgenerovaných bodů.

Odebrání bodu: Pro odebrání bodu nejprve dvojklikem označte příslušný bod. Poté stiskněte červené tlačítko s ikonou „×“, v horní liště aplikace. Bod bude automaticky odebrán včetně všech hran kostry, které do něj vedly.

Přechod k dalšímu kroku ověření

Po dokončení editace kostry podpisu lze přejít na další krok ověřování pomocí zeleného tlačítka s ikonou šipky v pravém dolním rohu aplikace. Samotný proces editace kostry je nepovinný a toto tlačítko se zobrazí uživateli ihned po načtení souboru s podpisem. Editaci tedy lze přeskočit a přistoupit ihned k dalšímu kroku ověření s kostrou, kterou vygenerovala aplikace.

5.5 Obrazovka výpočtu

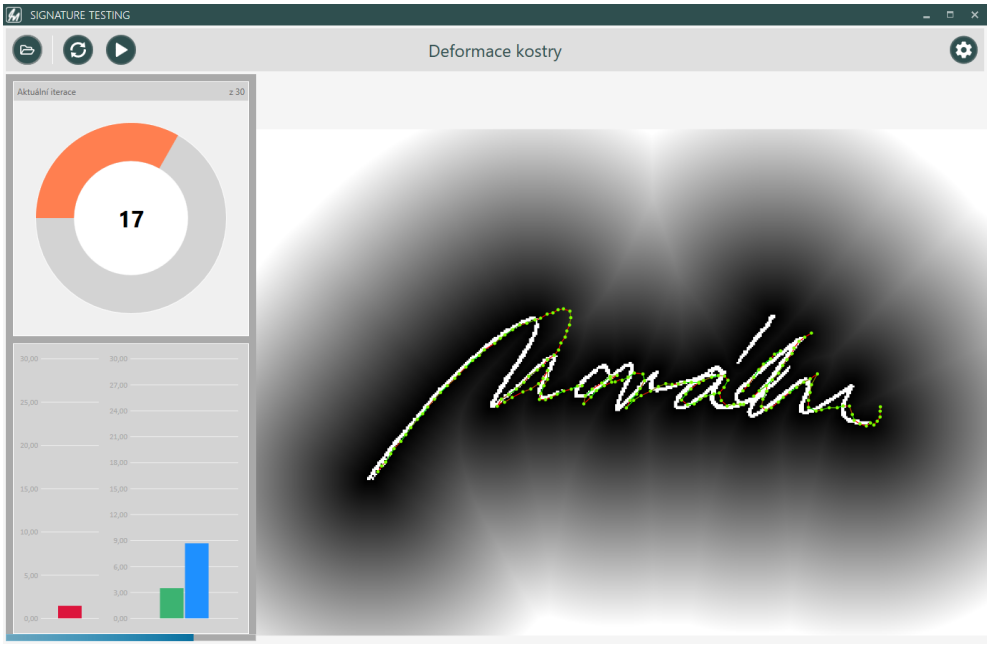
Po přechodu z editační obrazovky se uživateli zobrazí obrazovka výpočtu. Zde probíhá proces deformace kostry ověřovaného podpisu na podpis vzorový. Obrazovka slouží pro výběr vzorového podpisu, průběžné zobrazování procesu deformace uživateli a pro nastavení parametrů, s jakými bude proces deformace spuštěn.

Výběr vzorového podpisu

Po zobrazení se uprostřed obrazovky nachází šedé tlačítko s ikonou složky. Po stisku tohoto tlačítka se zobrazí dialog pro výběr vzorového podpisu. Aplikace očekává bitmapový obrázek ve formátu PNG nebo JPG o rozměrech 500×300 pixelů. Je vhodné, aby byla tato bitmapa již předzpracována a byly odstraněny tmavé pixely, které nesouvisí s linkou podpisu a narušovaly by tak výpočet mapy vzdáleností.

Po zvolení podpisového souboru bude podpis zpracován a spustí se výpočet jeho mapy vzdáleností. Tento výpočet může chvíli trvat. Výpočet skončí v momentě, kdy se na obrazovce zobrazí vzorový podpis s mapou vzdáleností zobrazenou odstupňovanými odstíny šedé barvy. Pokud chce uživatel zvolit jiný podpisový vzor, může tak učinit pomocí zeleného tlačítka s ikonou složky v levém horním rohu aplikace. Po jeho stisku se opět zobrazí dialog pro výběr souboru s podpisovým vzorem.

Po zobrazení vzorového podpisu se také vždy zobrazí kostra testovaného podpisu, kterou uživatel vytvořil v předchozím kroku.



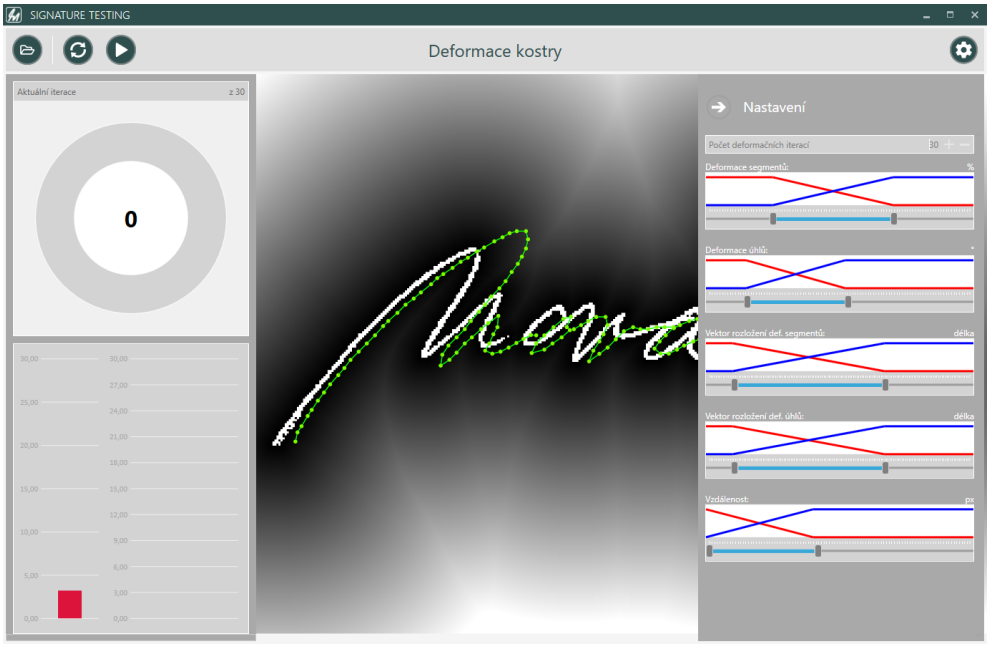
Obrázek 25: Obrazovka výpočtu

Nastavení výpočtu

Parametry deformačního procesu lze nastavit v panelu nastavení. Ten vyvolá uživatel stiskem tlačítka nastavení v pravém horním rohu obrazovky. Uzavření tohoto panelu provede uživatel opětovným stisknutím tlačítka nastavení nebo tlačítkem šipky v hlavičce panelu.

Počet deformačních iterací: V horní části panelu se nachází nastavení počtu iterací deformačního procesu, které budou provedeny. Tento parametr může uživatel nastavovat přímým vepsáním hodnoty do kolonky nebo pomocí tlačítek „+“ a „–“ vedle hodnoty.

Vyhodnocování pozice bodu kostry: Při deformačním procesu probíhá vyhodnocování nových pozic bodů na základě pěti měřených veličin popsaných v kapitole 3.5. Tyto veličiny jsou posuzovány vzhledem k jazykovým výrazům „malá hodnota“ a „velká hodnota“. Definice těchto hodnot lze v panelu nastavit pro každou veličinu samostatně. Ovládací prvek pro jejich nastavení se skládá ze dvou částí. V horní části se nachází grafické znázornění aktuálního nastavení obou jazykových hodnot k dané veličině. Červená linka znázorňuje malou hodnotu a modrá linka velkou hodnotu veličiny. Druhou částí je dvojitý posuvník, pomocí něhož lze nastavit začátek a konec sestupné a náběžné hrany. Při tažení levým i pravým posuvníkem se zobrazuje aktuální nastavovaná hodnota. Tahem za modrou linku mezi posuvníky lze posouvat oba posuvníky současně se zachováním délky intervalu mezi nimi.



Obrázek 26: Nastavení výpočtu

Potvrzení nastavení: Nastavení se projeví v programu automaticky ihned po změně hodnoty v panelu nastavení. Žádnou změnu nastavení tedy není nutné potvrzovat.

Spuštění výpočtu

Spustění procesu deformace provede uživatel stiskem zeleného tlačítka s ikonou ▶ v levé části horní lišty obrazovky. Následně se proces spustí, přičemž je po celou dobu procesu zobrazován aktuální stav kostry. Deformace jednotlivých segmentů kostry je znázorňována postupným přechodem zelené barvy na červenou, kde zelená znamená segment bez deformace a červená vyšší míru deformace (toto zabarvení neodpovídá přímo hodnotám velká a malá deformace zmiňované výše).

Levý boční panel: V levé části obrazovky výpočtu se nachází panel pro zobrazení průběžných informací o aktuálně probíhajícím deformačním procesu. V jeho horní části je graficky znázorňována aktuální iterace výpočtu a celkový počet plánovaných iterací. Ve spodní části se nachází sloupcový graf pro zobrazení aktuální míry deformace kostry a také průměrné vzdálenosti bodů kostry od vzorového podpisu. Graf nemusí zobrazovat nejaktuálnější hodnoty, protože jeho vykreslování je omezeno, aby nebyl zbytečně brzděn časově náročný výpočet procesu deformace.

Více deformačních procesů: Pro možnosti experimentování je uživateli umožněno spustit více deformačních procesů. Uživatel může navázat na předchozí deformaci dalšími iteracemi se stejným, nebo naopak pozměněným nastavením.

Dále má uživatel možnost vrátit kostru podpisu do původního stavu a provádět nové procesy deformace s různým nastavením. Krok vrácení kostry do původního tvaru provede uživatel stiskem zeleného tlačítka s ikonou šipek v levé části horní lišty obrazovky.

Přiblížení a pohyb:

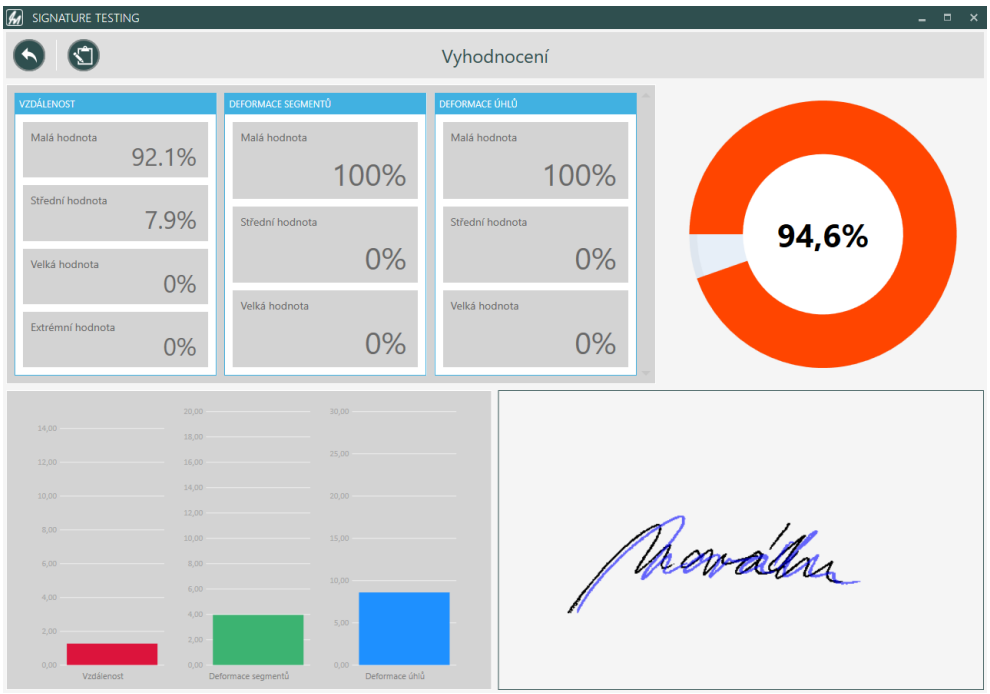
V průběhu zobrazení obrazovky výpočtu má uživatel po celou dobu možnost načtený podpis libovolně zvětšovat a oddalovat kolečkem myši, stejně jako v případě editoru kostry. Posouvání podpisu po pracovní ploše lze provést opět pravým tlačítkem myši. Uživatel tak může detailněji zkoumat stav kostry podpisu před, během a po procesu deformace.

Přechod k vyhodnocení ověřování

Po ukončení první deformace se zobrazí v pravém dolním rohu tlačítko s šipkou pro přechod k výslednému vyhodnocení ověření.

5.6 Obrazovka výsledků

Po přechodu z obrazovky výpočtu se uživateli zobrazí poslední obrazovka vyhodnocení výsledků. Zde jsou zobrazeny veškeré důležité informace získané v předchozích krocích ověření.



Obrázek 27: Obrazovka výsledků

Celkový výsledek

V pravé horní části obrazovky se zobrazuje výsledné procento, na kolik se aplikace domnívá, že autorem obou podpisů je stejná osoba. Hodnota je doplněna o její grafické znázornění nabývajícím kruhovým grafem.

Tabulka příslušnosti fuzzy množin

V levé horní části obrazovky se nachází tabulka příslušnosti jednotlivých deformačních veličin k fuzzy množinám inferenčního mechanizmu pro konečné výsledky. Každá z fuzzy množin má ve své buňce uvedeno procento, v jakém spadá aktuální hodnota veličiny do této fuzzy množiny.

Graf veličin deformace

V levé spodní části obrazovky se nachází graf zobrazující hodnoty výsledných deformačních veličin kostry, které byly vstupem pro celkové vyhodnocení.

Srovnání překrytím

V pravém spodním rohu obrazovky jsou zobrazeny oba porovnávané podpisy. Podpisy jsou umístěny přes sebe tak, jak byly vůči sobě vystředěny v procesu deformace. Tato část slouží pro vizuální porovnání obou podpisů s možností subjektivně zhodnotit kvalitu výsledku aplikace v kontextu očekávaných výsledků uživatelem.

Návrat zpět

Uživatel se může vrátit zpět na obrazovku výpočtu pomocí tlačítka s šipkou v levé části horní pracovní lišty. Po stisku se zobrazí obrazovka výpočtu s kostrou deformovaného podpisu ve stavu, v jakém byla před přechodem na obrazovku výsledků.

Spuštění nového testu

Uživatel spustí nové ověřování podpisů pomocí tlačítka umístěného vedle návratového tlačítka v levé části horní pracovní lišty. Po stisku se opět zobrazí prázdná obrazovka editoru kostry podpisu.

6 Výsledky aplikace

Po implementaci celé aplikace proběhlo její testování, ze kterého vzešlo několik bodů vhodných ke zmínění v této práci. Je zajímavé sledovat, jak různá nastavení deformačního procesu dávají velmi rozdílné výsledky a často jsou také značně závislé na konkrétních podpisech. Právě z tohoto důvodu byla do aplikace přidáná

možnost nastavení deformačního procesu, které je však přednastaveno výchozími hodnotami, která se během testování jevila jako nejlepší kompromis.

Pro jednoduché testovací linky funguje postup deformace velmi příznivě. Jejich kostry v naprosté většině během iterací dokonvergují ke vzorové lince nebo alespoň velmi blízko, jak jim to dovolí pružnost kostry. S narůstající složitostí linek ale také narůstá množství případů, kdy linky z různých důvodů konvergovat nemohou. V případě reálných podpisů jsou tyto situace dokonce celkem běžné.

Pro dva stejné podpisy aplikace vždy vrátí 100% shodu, což vyplývá už z postupu, jakým podpisy ověřujeme. Naopak pro zcela rozdílné podpisy vrací vždy nulové nebo velmi nízké procentuální hodnoty. Aplikace tedy v těchto hrubých obrysech funguje dobře. Problémy nastávají až u podpisů, které si jsou relativně podobné nebo se do značné míry překrývají v místech s velmi vysokou hustotou linek podpisu. Tabulky č. 2 a 3 obsahují výsledná data testování aplikace včetně výstupních veličin deformačního procesu. V tabulce 2 jsou obsažena data testování podpisů, při kterém byl testovaný podpis vždy pravý. Naopak tabulka 3 obsahuje výsledky testování falešných podpisů. Rozdělíme-li procentuální stupnici výsledků pomyslnou hranicí 50%, pak z dat v tabulkách výsledků vyplývá úspěšnost aplikace nad 90%. Je nutné dodat, že množina podpisů pro testování byla omezená a pro přesné statistické výsledky by bylo nutné provést testování nad objemnou množinou podpisů.

V této fázi vývoje zatím do procesu ověřování nevstupují žádná expertní data, na základě kterých by se zvyšovala úspěšnost ověření. Avšak i přes tento fakt má aplikace dostatečnou úspěšnost na to, aby principy ověřování v ní použité byly nasazeny jako automatický „hlídač“, který by upozorňoval na případné neshody podpisů v různých dokumentech. Na tato upozornění by poté mohli reagovat expertní pracovníci, kteří by provedli expertní analýzu.

Pozorované problémy aplikace

Největším pozorovaným problémem je situace, kdy body kostry testovaného podpisu leží blíže jiné linky, než ke které by tyto body měly deformací konvergovat. Body pak mají tendenci putovat právě opačným směrem, což má většinou za následek vysokou deformaci. Stejně tak většinou roste i vzdálenost, protože jiné části podpisu se nedostanou ke svým linkám kvůli omezené pružnosti kostry. Výsledky se potom pohybují kolem 50% shody nebo spíše negativně i případě stejného autora obou podpisů. Teoreticky bychom mohli uvažovat, že aplikace je pouze v tomto ohledu velmi přísná, protože i takováto odlišnost může být považována za důvodné podezření o nepravosti testovaného podpisu. Na druhou stranu u podpisů s vysokou hustotou linek je pravděpodobnost těchto jevů podstatně vyšší. Proto je tato vlastnost považována spíše za negativní jev, který by bylo vhodné co nejvíce potlačit.

Druhým problémem jsou „hrubé“ bitmapy podpisového vzoru. Pokud tato bitmapa obsahuje tmavé pixely kolem podpisu, které však nejsou součástí podspisových linek, pak tyto body zásadně ovlivní výpočet mapy vzdáleností defor-

Tabulka 2: Výsledky testování pravých podpisů

testovaný	vzorový	vzdálenost	def. segmentů	def. úhlů	výsledek
o1	o2	1,54	3,54	7,93	**82%**
o2	o3	1,33	3,24	9,96	**71%**
o3	o4	2,61	3,92	14,47	**43,9%**
o4	o5	2,68	4,24	8,92	**41,8%**
o5	o6	1,96	3,64	9,26	**75,4%**
o6	o7	1,2	3,45	11,59	**100%**
o7	o8	0,74	3,03	8,63	**100%**
o1	o3	0,47	2,83	7,61	**100%**
o2	o4	0,5	3,32	10	**75,2%**
o3	o5	1,09	2,75	7,56	**100%**
o4	o6	0,75	3,39	7,89	**100%**
o5	o7	1,19	3,65	8,72	**100%**
o6	o8	1,28	3,89	9,87	**75%**
m1	m2	2,84	4,97	12,56	**10,6%**
m2	m3	2,16	4,17	8,97	**71,3%**
m3	m4	1,01	3,65	10,18	**76,9%**
m4	m5	1,76	3,88	9,5	**77,1%**
m5	m6	0,91	3,33	9,17	**100%**
m6	m7	2,38	4,95	9,26	**48%**
m7	m8	1,58	4,16	10,29	**56,1%**
m8	m9	1,7	4,26	10,87	**48,3%**
m9	m10	1,35	3,42	8,85	**89,8%**
m10	m11	1,97	4,18	9,72	**73,3%**
m11	m12	1,49	3,58	9,01	**83,6%**
m12	m13	1,28	3,85	8,93	**93,9%**
m13	m14	0,78	3,21	9,11	**100%**
m14	m15	1,62	3,89	9,62	**77,5%**

mačního algoritmu a tím zcela ovlivní průběh výpočtu. Testovaný podpis těmito pixely reprezentujícími šum není příliš zatížený, protože jeho linky se používají pouze pro stavbu kostry a body, které jsou umístěny na tyto šumové pixely lze v editoru kostry odstranit. Nicméně je tento problém řešitelný i pro vzorové podpisy pomocí předzpracování bitmapy v grafickém editoru.

Vhodné směry dalšího vývoje

Aplikace poskytuje velkou a snadno rozšiřitelnou základnu pro další vývoj ve všech směrech tohoto tématu. Na základě prvotního vývoje lze stanovit několik směrů, kterými by měla případná další rozšíření nebo úpravy algoritmů aplikace pokračovat.

První možností dalšího pokračování této práce je potlačení výše zmíněného

Tabulka 3: Výsledky testování falešných podpisů

testovaný	vzorový	vzdálenost	def. segmentů	def. úhlů	výsledek
f-o10	o1	3,78	3,92	11,58	**0%**
f-o11	o2	2,88	4,06	10,14	**40,8%**
f-o13	o3	2,33	4,14	12,07	**48,3%**
f-o14	o4	3,13	4,14	11,95	**5%**
f-o15	o5	4,66	4,57	10,55	**0%**
f-o16	o6	15,73	4,3	9,56	**0%**
f-o17	o7	14,38	4,45	9,32	**0%**
f-o18	o8	24,81	4,04	7,08	**0%**
f-o19	o4	3,56	4,86	10,96	**0%**
f-o12	o5	3,54	4,44	11,96	**0%**
f-m1	m1	2,12	5,19	11,45	**25%**
f-m2	m2	1,43	3,81	10,14	**63,6%**
f-m3	m3	3,03	4,28	10,68	**19,1%**
f-m4	m4	2,05	5,02	11,84	**28,1%**
f-m5	m5	2,8	4,61	14,17	**24,3%**
f-m6	m6	3,51	4,37	10,58	**0%**
f-m7	m7	3,26	4,58	10,59	**0%**
f-m8	m8	4,64	4,26	16,88	**0%**
f-m9	m9	4,46	5,36	10,73	**0%**
f-m1	m11	2,92	7,48	11,42	**5,6%**
f-m2	m8	2,29	4,56	10,65	**24,8%**
f-m3	m15	3,12	4,25	12,1	**11,5%**
f-m4	m13	3,06	13,16	11,19	**0%**
f-m5	m10	2,58	5,48	16,9	**22,4%**
f-m6	m7	4,51	5,29	12,55	**0%**
f-m7	m8	4	5,14	10,27	**0%**
f-m8	m9	4,64	4,26	16,88	**0%**

problému, kdy body kostry leží blíže k jiné lince podpisu než ke které by měly skutečně konvergovat. Zde by bylo vhodné navrhnout postup, jakým přeskočit potřebnou vzdálenost správným směrem k lince podpisu. Pro aktuální verzi deformačního algoritmu se nabízí například testování vzdálenější pozice bodu od jeho aktuální pozice. Bohužel toto řešení nebude fungovat, protože s narůstající vzdáleností posunutí narůstá i deformace segmentů, což nedovolí dostatečný skok bodů.

V souvislosti s tímto problémem by se dala zvážit úprava vyhodnocovacího algoritmu jednotlivých pozic bodů kostry. Teoretickou možností, jak algoritmus upravit, by bylo přiřazení vah jednotlivým veličinám, které by bylo možné dynamicky měnit v závislosti na situaci, hodnotách jednotlivých veličin nebo třeba aktuální iteraci.

Dalším vhodným vylepšením je odstranění nebo alespoň částečné potlačení pevných křižovatek kostry podpisu. Snahou by tedy mělo být co nejvíce bránit vytváření bodů kostry s více než dvěma sousedy, kteří reprezentují protnutí dvou a více čar podpisu a tím vytvářejí pevné spojení těchto čar v jednom konkrétním bodě. Taková úprava by měla vést k menším deformacím kostry a lepšímu formování na vzorový podpis.

Čistě užitečným rozšířením aplikace by také byla implementace grafického editoru, který by uživateli dovolil odstranění šumu (černých pixelů, které nejsou součástí podpisu). Uživatel by tak nemusel provádět na vzorových podpisech grafické předzpracovávání, ale mohl by ho vyřešit v rámci této aplikace.

Závěr

Cílem této práce bylo implementovat a dále rozvíjet principy popsané José Vélezem, Ángel Sánchezem, Belén Morenem a José L. Estebanem v jejich článku z roku 2008 [3]. V průběhu času se ale tento úkol začal jevit jako velmi náročný. Návrh a následná implementace potřebných algoritmů byla velmi složitá a pro veškeré ladění a vývoj bylo vždy nutné implementovat grafické zobrazování všech dat, což byly také časově velmi náročné úkoly. Součástí práce bylo také vytvoření knihovny pro práci s fuzzy množinami a fuzzy inferenčním systémem, jejíž funkčnost musela být stoprocentně zaručena. Komplikovanost algoritmů, které tuto knihovnu využívají společně s velkým množstvím parametrů ovlivňujících výsledky algoritmů totiž znemožňuje odhalení jakýchkoliv chyb na nižší úrovni. V průběhu vývoje navíc docházelo k častým změnám funkcionality i datových struktur, protože se ukázaly pro další kroky ověřování nevyhovující a vývoj se tak vracel na začátek.

Vzhledem k celkové náročnosti zadání byla architektura aplikace navržena co nejvíce modulárně, aby bylo možné se k jednotlivým algoritmům vracet a přepracovávat je bez ovlivnění zbytku programu. Snahou bylo umožnit, aby na tuto práci bylo možné navázat v případě, že na konci vývoje některé části nebudou implementovány nebo budou jejich výsledky špatné. Vývoj ovšem nad očekávání dospěl do velmi příznivé fáze, kdy je aplikace schopna zpracovat testovaný i vzorový podpis, provést vlastní porovnání pomocí deformačního procesu a také vyhodnotit celkové výsledky, které jsou velmi uspokojivé vzhledem k prvotním skeptickým odhadům.

Toto téma bylo náročné nejen na obecné znalosti teoretické informatiky, ale také na schopnosti vývoje algoritmů, bez kterých by se tato práce neobešla. V předchozím textu jsou popsány některé problémy, jejichž vyřešení by značně posunulo kupředu úspěšnost výsledků této aplikace. Pevně doufám, že na tuto práci někdo naváže a pokusí se vyřešit zmíněné problémy, které již bohužel nebylo možné v této práci zpracovat. I přes náročnost problémů je zde popisovaný přístup ověřování podpisů nesmírně zajímavý a dává prostor pro vlastní návrh, experimentování a vývoj řešení.

A Obsah přiloženého CD/DVD

bin/
: Instalátor SETUP.EXE programu, spustitelné přímo z CD/DVD. Adresář obsahuje i všechny runtime knihovny a další soubory potřebné pro bezproblémový běh instalátoru z CD/DVD

doc/
: Text práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech příloh, a všechny soubory potřebné pro bezproblémové vygenerování PDF dokumentu textu (v ZIP archivu), tj. zdrojový text textu, vložené obrázky, apod.

src/
: Kompletní zdrojové texty programu SIGNATURE TESTING se všemi potřebnými (příp. převzatými) zdrojovými texty, knihovnami a dalšími soubory potřebnými pro bezproblémové vytvoření spustitelných verzí programu.

readme.txt
: Instrukce pro instalaci a spuštění programu SIGNATURE TESTING, včetně všech požadavků pro jeho bezproblémový provoz.

Navíc CD/DVD obsahuje:

data/
: Ukázkové a testovací podpisy použité v práci a pro potřeby testování práce při tvorbě posudků a obhajoby práce.

install/
: Instalátory aplikací, runtime knihoven a jiných souborů potřebných pro provoz programu SIGNATURE TESTING.

literature/
: Vybrané položky bibliografie, příp. jiná užitečná literatura vztahující se k práci.

Literatura

[1] BENEŠ, Bedřich; FELKEL, Petr; SOCHOR, Jiří; ŽARA, Jiří : *Moderní počítačová grafika*, Druhé vyd. 2008.

[2] DOOLEY, Laurence S.; HOWING, Frank; WERMSER, Diederich : *Linguistic contour modelling through Fuzzy Snakes*, Elektronická publikace, 1999. Dostupné z: www2.ostfalia.de/export/sites/default/de/pws/wermser/veroeffentlichungen/material/cimca99.pdf

[3] ESTEBAN, José L.; MORENO, Belén; SÁNCHEZ, Ángel; VÉLEZ, José : *Fuzzy shape-memory snakes for the automatic off-line signature verification problem*, Elektronická publikace, 2008.

[4] GONZALEZ, Rafael C.; WOODS, Richard E. : *Digital Image Processing*, Třetí vyd. 2008.

[5] MODRLÁK, Osvald: *Fuzzy řízení a regulace*, Elektronická publikace, 2002. Dostupné z: https://www.kirp.chtf.stuba.sk/ bakosova/wwwRTP/tar2fuz.pdf

[6] ŠPANĚL Michal: *Obrazové segmentační techniky*, Elektronická publikace, 2005, Dostupné z: http://www.fit.vutbr.cz/ spanel/segmentace